NETVIBES | Exalead CloudView

CloudView CV23

Connector Programmer

Table of Contents

Connector Programmer

This guide explains how to develop, deploy, and configure Exalead CloudView custom Java or .NET connectors using the Push API. This public document API allows you to index data from any source with Exalead CloudView.

Audience

This guide is mainly destined to software programmers or users with a few programming skills.

It is assumed that the reader has experience in the operating system on which the Exalead CloudView server is installed.

Accessing the Push API

Access Push API with	Description
Java SDK	Java Clients SDK is delivered in the Exalead CloudView kit in `<INSTALLDIR>/sdk/java-clients`. This SDK contains all required material to develop external applications interacting with Exalead CloudView. It includes the following content: • `/docs` - API documentation (javadoc); also available online at Exalead CloudView Public APIs Java SDK • `/lib` - the jars to use • `/samples` - code samples
.NET SDK	.NET Clients SDK is delivered in the Exalead CloudView kit in `<INSTALLDIR>/sdk/dotnet-clients`. This SDK contains all required material to develop external Push API applications interacting with Exalead CloudView. It includes the following content: • `/docs` - API documentation • `/lib` - the library including the `dll` files • `/samples` - code samples for the Search API and the Push API
Raw access	The default endpoint for the Push API is: `http://<HOSTNAME>:<BASEPORT+2>`

Further Reading

You might need to refer to the following guides:

Guide	for more details on
Connectors	standard connector's configuration.
Mashup Programmer	Mashup UI customization.
Programmer	Exalead CloudView customization.
javadoc	Java methods and classes. Available in the CloudView Public APIs Java SDK.

What's New?

There are no enhancements in this release.

About the Push API

The Push API supports the basic operations required to develop new connectors, both managed and unmanaged.

What is the difference between a managed and unmanaged connector?

- A managed connector is a piece of code running within Exalead CloudView. It must be packaged as a Exalead CloudView Plugin to be deployed and configured in Exalead CloudView. You must develop it in Java, using the Connectors Framework API available in:
  - `<INSTALLDIR>\sdk\java-customcode` for V6R2014 and higher versions.
  - `<INSTALLDIR>\sdk\cloudview-sdk-java-connectors` in previous versions.
- An unmanaged connector is an external component that sends data to Exalead CloudView using the Push API. You can develop an unmanaged connector in any language, either by using Exalead CloudView Push API clients (available in Java, C# and PHP), or by directly targeting the HTTP API. You must manage and deploy unmanaged connectors yourself, as Exalead CloudView is not aware of these connectors.

All standard Exalead CloudView connectors are managed connectors. For more details, see the Exalead CloudView Connectors Guide.

What are the goals of a connector?

A connector can be seen as a portal between two worlds, Exalead CloudView's index, and a specific data source.

This portal is used at two times:

- At Indexing Time
  - Full indexing: Captures a snapshot of the data source's current state in Exalead CloudView.
  - Incremental indexing: Synchronizes modifications made on the data source with the Exalead CloudView index.
- At Search Time
  - The user performs a search in Exalead CloudView.
  - A first check is made between the indexed document security tokens and the user's security tokens. For more information, see Developing a Security Source.

- Exalead CloudView only displays the list of authorized documents matching the user query.
- A second check is made between the document security tokens in the data source and the user's security tokens, when documents are fetched (downloaded or previewed). This is done through the `getDocumentSecurityTokens` method.

Push API concepts

Documents

Documents can be defined as all the objects to be indexed by Exalead CloudView, regardless of file or entity type in the data source. For example, HTML, JPG or CSV files, database records are all considered documents within Exalead CloudView, since they are all converted into a Exalead CloudView-specific document format (also known as a PAPI document) after being scanned by a connector.

Items are the objects to be indexed by Exalead CloudView, regardless of file or entity type in the data source. For example, in OnePart, 3D CAD files, JPGs, PDFs are all considered items in the index.

A PAPI Document is an exchange format between the connectors and Exalead CloudView. It's an abstraction, so that all connectors speak the same language to the index. The Push API handles documents that contain the following elements:

- URI
- Stamp (optional)
- Metas
- Parts
- Directives

URI

URI is the unique identifier of the document inside the indexed corpus of the connector.

Note: The "URI" described in this document is an opaque string (with optional "/" character hierarchy), and is NOT necessarily a "URI" as per RFC 2396, even if connectors may use regular Internet URI. For example:

Sample URI	Interpreted as
a/b/doc	Folder: a

Sample URI	Interpreted as
	|_ Folder: b |_ Document: doc
a/b///doc	Folder: a |_ Folder: b |_ Folder: (empty name) |_ Folder: (empty name) |_ Document: doc

Stamps

Stamp is a fingerprint that represents the "state", or "version" of the document. Stamps are stored by Exalead CloudView, and retrieved back by the connector to determine which version of the document has been indexed, and whether it should be updated. The document will be updated if the new stamp is not equal to the previous one.

See also the Stamp-based synchronization.

Meta

Document metas, not to be confused with hit metas, are pieces of text belonging to a document that have associated values, such as title or size. Document metas are stored either as an index field or as a category. Context is sometimes used as a synonym for document meta.

Parts

Parts represent the binary parts of the content to be converted and indexed like a file. Usually, only one part is needed, but you may need to link some attachments to the content. All parts are merged together and are associated to the same URI.

Note that:

- A PAPI `Part` has a name (in all Exalead CloudView versions)
- The default `Part` name is `master`
- There must be one master `Part` per PAPI document (for preview)

Thus when a PAPI document has several parts:

- They must all have different names
- One of them must be named `master` – to set it you can use the `com.exalead.papi.helper.part.setAsMaster()` member method.

The `Part` name can be set with the following member methods:

```
com.exalead.papi.helper.Part(String name, bytes[])com.exalead.papi.helper.Part.setNam
```

Directives

Directives are internal properties embedded in a Exalead CloudView document. They specify either orders on how to treat the document, or information on how to index the document.

Some directives are available at document level:

- `datamodel_class`: determines the data model class of the document. If this directive is not found, the data model class specified in the source connector configuration will be used. If the source connector does not have a class, we use the data model default class. For example:

  ```
  final Document myDocument = new Document("docId");
  myDocument.setCustomDirective("datamodel_class", "myDocumentClass");
  ```

- `forcedSlice`: overrides the automatic load balancing of documents in the Exalead CloudView slices, by forcing the slice on which documents will be stored.
- `sameSlice`: (for V6R2014 and higher) forces the document to use the slice of another document by specifying the URI of this document.

Some directives are available at the part level to help the converter determine the content type. Note that the values of these directives cannot be null. Examples of supported directives:

- `filename`: the filename of the document
- `mimeHint`: the hint mime parameter
- `mime`: the forced mime (use with caution)
- `encoding`: the encoding of the document

The analysis pipeline takes both metas and directives into account to determine how to process a document. For example, to get the file name of a document part, it looks for both the `file_name` meta and the `filename` directive, if any. We recommend using the meta when data must be indexed.

Important: When there are several directives in a document, delete operations are processed BEFORE add operations.

Consolidation Server directives

The following table shows the hard-coded order of Consolidation Server directive operations. These directives are created automatically by the Consolidation Server when you push methods to the transformation processors.

To add these directives, you can (using `com.exalead.cloudview.consolidationapi.PushAPITransformationHelpers`)

- Include them directly within your documents.
- or use pre-aggregation transformation rules in the Exalead CloudView **> Consolidation config**.

  Note: For more details on the Consolidation Server, see the Consolidation Server Guide.

Checkpoints

Checkpoints are opaque string, used by connectors for synchronization purposes. For more details, see Checkpoints and Checkpoint-based synchronization.

Synchronization

After it has sent all documents from a datasource to Exalead CloudView, a connector must generally keep the index up to date. This process is called synchronization. You can use either stamp-based or checkpoints-based synchronization to synchronize a data-source. For more details, see Implementing Synchronization.

Supported Text Encodings

Parts binary content in text MIME subset may have use any recognized encodings (see the list of available encodings below). The proper encoding should be filled in the part meta-data (encoding or encoding hint).

All other concepts shall only use UTF-8 (or its 7-bit restriction ASCII) as sole encoding, especially all Push API multipart commands.

Push API HTTP Level

The Push API is a simple HTTP API. The API user can implement a Push API client using either the language of choice, or the client-side wrappers for the API. The supported languages are C# and Java.

Note: While the exact syntax for the chosen language will differ, the recommendation is that the method and arguments should have the same names.

Push API at the HTTP level

HTTP command parameters

Parameters can be sent to the server in different ways using either `[URL]` or `[FORM]`.

Parameter	Description
`URL`	The parameter in the request URL, for example: `...../addDocument?uri='file://mydir/file1.doc'`
`FORM`	The parameter is part of the form data

The required way is specified in the command description.

HTTP methods

The HTTP methods used are the following:

Parameter	Description
GET	The GET method
POST	The POST method can be encoded as: ?multipart/form-data content-type (RFC 2388), or application/x-www-form-urlencoded ?application/octet-stream (for xxx_monopart commands)

This document describes the HTTP POST method.

HTTP encoding

When dealing with text (for example, metadata key or values), the only accepted encoding is UTF-8. No other encoding is supported.

HTTP command response

The processing of HTTP Push API operations may be asynchronous. This means that requested add or delete operations are accepted but we do not know exactly when they will be performed. However, errors may occur at a lower level, so here is a description of the default HTTP responses.

HTTP Response	Description
`OK (200)`	No problem during parameters de-serialization process.
`NO_CONTENT (204)`	No content.
`20X`	Helpers should consider 204 and all 20X statuses as OK.
`BAD_REQUEST (400)`	An error occurred while parameters were parsed or during command treatment. The body of the result contains the error description (see below for the xml format of the error description).
`METHOD_NOT_ALLOWED (405)`	The use of the POST and GET methods is strict with the HTTP Push API. Only the specified methods are authorized for each command.
`UNAUTHORIZED (401)`	The access to connector operations through HTTP is protected using basic authentication, and has been forbidden.
`INTERNAL_ERROR (500)`	An unexpected error occurred on the server side.

HTTP time out

The HTTP `TimeOut` should be set to `infinite` in the event of the server being busy at request time. This prevents the connector from retrying to connect to the server.

Push API Client Implementation Recommendations

When you implement a Push API client (for example, for a language not currently supported) you will use HTTP API calls to create all the Push API client methods. For example, the add and enumerate methods.

Conventions to follow

You should follow certain conventions for method interface and method names:

- The interface should be called `PushAPI`. For example, `PushAPI`(Java), `IPushAPI`(C#)…
- The `HttpPushAPI` class implements `PushAPI`. The constructor must be in the form:

```
HttpPushAPI(PushAPIVersion version, String host, String port, String connectorName
String connectorType, String login, String password)
```

- The `FakePushAPI` class implements `PushAPI`. This class is optional and should be a simulator for test.
- The `PushAPIFactory` class should be implemented to create `HttpPushAPI` and `FakePushAPI`
  - `PushAPI createFake()` (optional)
  - `PushAPI createHttp(PushAPIVersion version, String host, String port, String connectorName, String connectorType, String login, String password)`

The following description is based on C# naming conventions.

Argument	Description
`host`	The Push API server host name.
`port`	The Push API server port number (by default, `<BASEPORT> + 2`).
`PushAPIVersion`	The Push API version. For the moment, the only accepted value is `PAPI_V4` for Exalead CloudView 5.x and 6.x For example, `PushAPIVersion.PAPI_V4`
`connectorName`	The name of the connector.
`connectorType`	The type of the connector. It is mainly used for license checking purpose (the available connector types are described in your product license).
`login`	The basic authentication login string. If not null, the basic authentication mode will be enabled.
`password`	The basic authentication password string. If not null, the basic authentication mode will be enabled. Login and Password must either be both null or not null.

Methods

Each class must implement specific methods.

Note: These methods are described using Java naming conventions. They must be adapted to the language of choice.

Error messages and exceptions

The Push API client should use an asynchronous mechanism while treating add and delete operations. This means that requested add or delete operations are accepted immediately but will be executed later. Every method of the Push API client should send exceptions as follows:

- `ServerUnavailableException` – when the remote Exalead CloudView product cannot be reached (network or overload errors).
- `InvalidConnectorException` – if the Exalead CloudView Push API server does not recognize the client Connector name.
- `UnknownErrorException` – when an unexpected error occurs.

Additionally, we recommend implementing optional exceptions. List of possible error types:

- `InvalidConnectorNameError`
- `InvalidParameterError`
- `UnknownError`

For languages that do not support the exception mechanism, the language error management should be used instead. The convention for error serialization is the following:

```
<error>
 <type> ... </type>
    <short_message>  ... </short_message>
    <message>  ... </message>
</error>
```

Operational status

Because some operations on documents and checkpoints are performed asynchronously (add, delete, set checkpoint), it's important to know at which state of the processing flow the command is performed.

Operations can have the following processing statuses:

Status	Description
`Pushed`	The operation has been received by Exalead CloudView, but can be lost if a crash occurs before the sync. All add, delete, and set checkpoints (`Add()`, `Delete()`, `SetCheckpoint()`) operations create a document in the `Pushed` state.
`Synced`	The operation has been synced to disk, to guarantee crash-proofness. The operation cannot be lost anymore. When you synchronize explicitly or choose to synchronize when you set a checkpoint, all documents and checkpoint operations are synced to disk.
`Searchable`	The operation has been propagated to the index, and documents can be found at search time.

The following methods operate on documents in specific states:

Methods	operate on documents with statuses...
Enumerating checkpoints and synced entries, getting checkpoints and counting synced entries: `EnumeratesSyncedEntries()` `GetCheckpoint()` `EnumerateCheckpointsInfo()` `countSyncedEntries()`	`synced` and `searchable`.
Getting document status and clearing checkpoints: `GetDocumentStatus()` `ClearAllCheckpoints()`	`pushed`, `synced`, and `searchable`.

Push API Client Methods

This section describes the Push API client methods (using Java conventions) to implement with the corresponding HTTP Push API POST methods.

void ping()

void startPushSession()

void stopPushSession()

void addDocument(Document document) and void addDocumentList(Document[ ] documentList)

void updateDocument(Document document, string[] fields) and void updateDocumentList(Document[ ] documentList, string[][] fieldsList)

void deleteDocument(String uri) and void deleteDocumentList(String[] uris)

void ping()

This method tests the connection with the server for the specified `connectorName`. This test should be called after the construction of the Push API.

The purpose of this method is to:

- test the server availability
- check for the existence of the `connectorName` and its security
- compare the PAPI Versions `X-Papi-Version`

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP method

The method used is:

```
GET http://<host>:<port>/papi/4/connectors/<connectorName>/ping
```

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

void startPushSession()

This method is used to start a new PushAPI session. This allows you to handle a "session" when working with the Push API server.

It aims to solve the following use case:

- the connector starts an indexing phase, and starts sending documents to the Push API server,
- the Indexing Server crashes or is being killed (or the server suddenly reboots); documents previously received are lost,
- the Indexing Server restarts,
- the connector sends remaining documents to the Push API server, unaware that the remote Push API server died, and the synchronization is therefore in a “lost” state.

This is done by introducing a session identifier (an integer) that identifies the remote component pushing the documents - this identifier changes each time the Exalead CloudView session manager (re)starts.

HTTP method

The `get_current_session_id` command allows you to get the remote Push API server session ID, which is generated when the Push API server starts. The method used is:

```
GET http://<host>:<port>/papi/4/connectors/<connectorName>/
get_current_session_id
```

HTTP response

The `get_current_session_id` command returns the current Push API server session id (long integer ; at least 63-bit). This identifier is used internally by Push API client helpers.

API response

The API function does not return any value. It throws a `PushAPISessionExistsException` if a session is already opened.

void stopPushSession()

This command is used to stop a PushAPI session previously opened by `startPushSession` and clears the internal session id. This command has no parameters.

HTTP response

No corresponding HTTP request exists for this client function.

API response

It throws a `PushAPISessionNotFoundException` if no session was opened.

void addDocument(Document document) and void addDocumentList(Document[ ] documentList)

This method requests to add a document. If a document with the same URI has already been added, the document will be updated.

Note: If the conversion of a Part fails, this Part is not indexed but the other Part and the Metas are included.

Document data types

When you implement the `addDocument` method you must send one or more documents to be added to the index. The `Document` object should contain:

Types	Description
`uri`	A URI, which is an opaque string that uniquely identifies the document from the connector point of view. See also URI.

Types	Description
`stamp`	An optional Stamp, which is an opaque string that the connector may use to track document changes. Document stamps may be retrieved through the `getDocumentStatus` method. See also Stamps.
`MetaContainer`	The `MetaContainer` of the document. Metadata are open name-value pairs. For a complete list of metadata understood by the API, see Metadata Examples.
`PartContainer`	The `PartContainer` of the document. The Connector sends raw bytes containing the document content. Exalead CloudView conversion services will translate and extract the textual content of the document before indexing. The Part contains a `DirectiveContainer`.
`DirectiveContainer`	The `DirectiveContainer` of the document (different from the directive associated to a Part).

Implement the part object

The `Part` object must provide accessors for the following predefined directives:

- `encoding`
- `filename`
- `mimeHint`
- `certifiedMime`

To set a custom directive, the `Part` object must also provide a method, for example:

```
public void setCustomDirective(string name, string value)
public void setCustomDirective(Directive directive)
public void setCustomDirective(string name, string[] values)
public void addCustomDirective(string name, string value)
```

Implement the document object

The `Document` object must provide accessors for these predefined directives:

- `forcedSlice`
- `sameSlice`

And a method to set a custom directive:

```
public void setCustomDirective(string name, string value)
public void setCustomDirective(Directive directive)
public void setCustomDirective(string name, string[] values)
public void addCustomDirective(string name, string value)
```

HTTP parameters

The `add_documents` parameters are described in the table below.

Important: This method sends HTTP POST requests.

Note: These parameters must be repeated (with a different `id`) for every document you want to send.

For better performance, we recommend using a `multipart/form-data` instead of `application/x-www-form-urlencoded`.

Parameter	Location	Description
`PAPI_<id>:uri`	`[URL/FORM]`	The `uri` parameter is the string of the document URI.
`PAPI_<id>:stamp`	`[URL/FORM]`	The optional `stamp` parameter is the string representing the document's Stamp.
`PAPI_<id>:meta:<meta_name`	`[URL/FORM]`	The `meta_*` parameter is a string containing the value of the metadata referenced by metaname. Multiple values may exist for the same parameter. You must generate as many parameters as there are values.
`PAPI_<id>:directive:<directive_name>`	`[URL/FORM]`	The list of optional supported directives (at the document level): • `forcedSlice`: advanced feature
`PAPI_<id>:part_bytes:<par`	`[URL/FORM]`	The `part_bytes` parameter is the content of the document's part that is identified by `part_name`.
`PAPI_<id>:part_directive:<part_name>:<directive_na`	`[URL/FORM]`	The list of optional supported directives (at the part level): • `filename`: the document filename • `mimeHint`: the hint mime parameter • `mime`: the forced mime (use very carefully)

Parameter	Location	Description
		• `encoding`: the document encoding
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

void updateDocument(Document document, string[] fields) and void updateDocumentList(Document[ ] documentList, string[][] fieldsList)

There are two update methods in the PushAPI: `updateDocument(Document doc, String[] fields)` and `updateDocumentList(Document[] docList, String[][] fieldsList)`

The first one is used to update one document, the second one to update several documents at once. The `fields`/ `fieldsList` parameters are not handled yet, so let's say they are useless as for now.

To update a document, you have to call one of these methods with a new document which:

- has the same URI as the one you want to update,
- and contains the updated parts/ metas.

The parts/ metas that do not have to be updated will be fetched from the document cache, so there is no need to put them in the document used for update.

Constraints

- For the update feature to work, you must either enable the Build Group document cache or target another Consolidation Server. For more information, see "Using Document Cache" in the Exalead CloudView Connectors Guide.
- Only documents that have been added after the document cache has been enabled will be updatable.

Notes

- The old values of multivalued metas will be dropped. If you want to update a multivalued meta by adding values, you have to put the old values you want to keep in the document used for update too.
- Remember that parts = fields and metas = fields. The index fields that will be updated depend on the part/meta field mappings, not on the part/meta names. For example, if you want to update the "`text`" field, you probably want to put an updated "`master`" part in the document used for update, and not a "`text`" meta.
- The document in the document cache is updated too, so subsequent updates of a document do not need to be cumulative.
- It is a good idea to perform batches of updates instead of single updates.

Document data types

When you implement the `updateDocument` method you must send one or more documents to be updated to the index. The `Document` object should contain:

Types	Description
`uri`	A URI, which is an opaque string that uniquely identifies the document from the connector point of view. See also URI.
`stamp`	An optional Stamp, which is an opaque string that the connector may use to track document changes. Document stamps may be retrieved through the `getDocumentStatus` method. See also Stamps.
`MetaContainer`	The `MetaContainer` of the document. Metadata are open name-value pairs. For a complete list of metadata understood by the API, see Metadata Examples.
`PartContainer`	The `PartContainer` of the document. The Connector sends raw bytes containing the document content. Exalead CloudView conversion services will translate and extract the textual content of the document before indexing. The Part contains a `DirectiveContainer`.

Types	Description
`DirectiveContainer`	The `DirectiveContainer` of the document (different from the directive associated to a Part).

Implement the part object

The `Part` object must provide accessors for the following predefined directives:

- `encoding`
- `filename`
- `mimeHint`
- `certifiedMime`

To set a custom directive, the `Part` object must also provide a method, for example:

```
public void setCustomDirective(string name, string value)
public void setCustomDirective(Directive directive)
public void setCustomDirective(string name, string[] values)
public void addCustomDirective(string name, string value)
```

Implement the document object

The `Document` object must provide accessors for these predefined directives:

- `forcedSlice`
- `sameSlice`

And a method to set a custom directive:

```
public void setCustomDirective(string name, string value)
public void setCustomDirective(Directive directive)
public void setCustomDirective(string name, string[] values)
public void addCustomDirective(string name, string value)
```

HTTP parameters

The `update_documents` parameters are described in the table below.

Note: These parameters must be repeated (with a different `id`) for every document you want to send.

For better performance, we recommend using a `multipart/form-data` instead of `application/x-www-form-urlencoded`.

Parameter	Location	Description
PAPI_<id>:uri	[URL/ FORM]	The uri parameter is the string of the document URI.
PAPI_<id>:stamp	[URL/ FORM]	The optional stamp parameter is the string representing the document's Stamp.
PAPI_<id>:meta:<meta_name	[URL/ FORM]	The meta_* parameter is a string containing the value of the metadata referenced by meta_name. Multiple values may exist for the same parameter. You must generate as many parameters as there are values.
PAPI_<id>:directive:<directive_name>	[URL/ FORM]	The list of optional supported directives (at the document level): forcedSlice: advanced feature
PAPI_<id>:directive:field	[URL/ FORM]	Not supported for the moment.
PAPI_<id>:part_bytes:<par	[URL/ FORM]	The part_bytes parameter is the content of the document's part that is identified by part_name.
PAPI_<id>:part_directive:<part_name>:<directive_na	[URL/ FORM]	The list of optional supported directives (at the part level): filename: the document filename mimeHint: the hint mime parameter mime: the forced mime (use very carefully) encoding: the document encoding
PAPI_session	[URL]	The optional parameter that retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

void deleteDocument(String uri) and void deleteDocumentList(String[] uris)

Request to delete a document on the specified URI list.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/delete_documents
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
PAPI_uri	[URL]	The uri parameter is the string of the document URI. To delete many files, send multiple PAPI_uri parameters.
PAPI_sessio	[URL]	The optional parameter that retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

However, no exception or error message is reported if the URI is unknown or refers to a document that was already deleted.

void deleteDocumentsRootPath(String rootPath [, Boolean recursive=true])

Deletes a set of documents (collection) specified by a rootPath. It is possible to only delete documents at the first level of the rootPath (not recursively) by using the recursive flag.

Data types

The object contains:

Types/flag	Description
`rootPath`	A part of the URI used to select a subset of the corpus. If the `rootPath` value is an empty string ("") then the whole collection will be deleted. Note that the `rootPath` means that the beginning of the URI must match. See also URI.
`recursive`	The recursive flag indicates that the deletion also impacts subfolders.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/delete_documents_root_pat
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
`PAPI_rootPath`	`[URL]`	The `rootPath` parameter is the string representation of the `rootPath`. It can take the form: `/root/subdir1/subdir2/subdir3/subdir3/...`
`PAPI_recursiv`	`[URL]`	A boolean representation of the flag: '`true`' for true, '`false`' for false.
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

However, no Exception or error message is returned if the `rootPath` refers to an empty (inexistant) subset of the corpus.

DocumentStatus getDocumentStatus(String uri) and DocumentStatus[] getDocumentStatusList(String[] uriList)

This method retrieves the status of a document within the indexed corpus specified by the `URI` parameters.

This status may be used by the connector to compare with the document status in the source, and then determine whether the document needs to be updated. The structure is serialized and returned in the response body.

The `getDocumentStatusList` method retrieves the status of a list of documents within the pushed corpus.

Data types

The `DocumentStatus` object contains:

Types	Description
`uri`	A URI is an opaque string that uniquely identifies the document from the connector point of view. See also URI.
`stamp`	An optional Stamp. See also Stamps.
`exist`	A boolean that indicates the indexing status of the document: • `true` indicates that a document with the given uri has already been sent to the Indexing System. However, this does not guarantee that the document has been indexed nor that the document can be seen by the user. • `false` indicates that the given uri is unknown to the Indexing System.

```
class DocumentStatus
{
  String getUri();
  String getStamp();
  boolean isExist();
}
```

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/get_documents_sta
```

HTTP parameters

The HTTP parameters are described in the table below.

Parameter	Location	Description
`PAPI_uri`	`[URL]`	The `uri` parameter is the string of the document URI. To delete many files, send multiple `PAPI_uri` parameters.

Parameter	Location	Description
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

If successful (`status = OK`), then the body contains the serialized form of the `DocumentStatus` in XML format.

Here is the response format for each entry:

```
[M/D] [space] [url_encode(URI)]          [space] [escape(STAMP)] [\n]
\ only if document is existing /
```

Where:

- `url_encode()` – is a function which performs an url encoding of the given value.
- `escape()` – is a function which replaces `\r` and `\n` with `\\r` and `\\n`.
- `M/D` – `M` indicates a missing entry, `D` indicates an existing document.

ulong setCheckpoint(String checkpoint [, String name] [, sync=false])

The `setCheckpoint` method sets checkpoints in the indexing system. If the optional name is specified, then the related checkpoint is changed.

Checkpoints are used when:

- The connector must process a journalized or logged data source, which can be abstractly represented as a flow of "add" and "delete" events in the corpus, and where an id can be used to refer to events on a timeline. The connector will then call the `setCheckpoint` command from time to time, with the id referring to the last add or delete events which have been sent to the Indexing System.
- Crash-proof synchronization is required. Upon crash, or system restart, the connector will call the `getCheckpoint` method to retrieve the last checkpoint saved by the Indexing System. The Indexing System guarantees that any add or delete commands called before that checkpoint were saved and will never be lost.
- To keep track of the synchronization.

The optional parameter name can be used if many checkpoints are needed for a given source. Default value is `""`.

The `sync` flag can be used to force the sync of the pending operations before returning control. Once synced, the document is pushed and securely handled by Exalead CloudView.

The `setCheckpoint` method returns the serial of the last pending operation before the checkpoint. It could be used to check when this document is indexed and searchable.

Note: A `getCheckpoint()` called immediately after a `setCheckpoint()` set with the sync parameter to `false` may not return the last value. `getCheckpoint()` always returns the last synced checkpoint.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/set_checkpoint
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
`PAPI_checkpo`	`[URL]`	The `checkpoint` parameter is the string of the checkpoint value.
`PAPI_name`	`[URL]`	This optional parameter can be used when you need to manage many checkpoints for a connector.
`PAPI_sync`	`[URL]`	The `sync` parameter is the string representation of the sync's value. If true, it triggers a `sync` operation.
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard responses. See HTTP command response.

If successful (`status = OK`), then the body contains the serialized form of the serial, which is the string value of the serial.

String getCheckpoint([String name])

The `getCheckpoint` method retrieves checkpoints in the indexing process.

The optional parameter name can be used if many checkpoints are needed for a given source. The default value is "".

A `getCheckpoint()` called immediately after a `setCheckpoint()` set with the sync parameter to `false` may not return the last value. `getCheckpoint()` always returns the last synced checkpoint.

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/get_checkpoint
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
`PAPI_name`	`[URL]`	This optional parameter can be used when you need to manage many checkpoints for a connector.
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

If successful (`status = OK`), then the body contains the serialized form of the checkpoint, which is the string value of the checkpoint.

String getCheckpoint([String name, Boolean showSynchronizedOnly])

This `getCheckpoint` method retrieves checkpoints in the indexing process.

The `name` parameter corresponds to the checkpoint name. The default value is "".

If the `showSynchronizedOnly` parameter is set to `false`, you will see all checkpoints, even those that are not yet synchronized to disk. If set to `true`, you will see only synchronized checkpoints.

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/get_checkpoint_in
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
PAPI_name	[URL]	This parameter is used when you need to manage many checkpoints for a connector.
PAPI_showSynchror	[URL]	This parameter is used to specify if you want to retrieve synchronized checkpoints only.
PAPI_session	[URL]	This optional parameter retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

If successful (status = OK), then the body contains the serialized form of the checkpoint, which is the string value of the checkpoint.

void clearAllCheckpoints()

The clearAllCheckpoints method is used to reset all checkpoints values, including the checkpoints with optional names.

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
PAPI_session	[URL]	The optional parameter that retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/clear_all_checkpoints
```

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

CheckpointsInfoIterator enumerateCheckpointsInfo()

Opens an Iterator over the list of defined checkpoints. Iterated results are streamed and used when needed.

The default checkpoint has the name `""` (empty string).

Data types

A `CheckpointsInfoIterator` is an abstract object used to retrieve `CheckpointsInfo`.

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/enumerate_checkpo
```

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

Here is the response format for each entry:

```
[url_encode(NAME)] [space] [escape(VALUE)] [\n]
```

Where:

- `url_encode()` – is a function which performs an url encoding of the given value.
- `escape()` – is a function which replaces `\r` and `\n` with `\\r` and `\\n`.
- `NAME` – can be empty.

CheckpointsInfoIterator enumerateCheckpointsInfo (boolean showSynchronizedOnly)

Opens an Iterator over the list of defined checkpoints, with a boolean parameter allowing to retrieve either synchronized checkpoints only (true) or all checkpoints (false). Iterated results are streamed and used when needed.

The default checkpoint has the name `""` (empty string).

Data types

A `CheckpointsInfoIterator` is an abstract object used to retrieve `CheckpointsInfo`.

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
`PAPI_showSynchror`	`[URL]`	This parameter is used to specify if you want to retrieve synchronized checkpoints only.
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/enumerate_stated_
```

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

Here is the response format for each entry:

```
[url_encode(NAME)] [space] [escape(VALUE)] [\n]
```

Where:

- `url_encode()` – is a function which performs an url encoding of the given value.
- `escape()` – is a function which replaces `\r` and `\n` with `\\r` and `\\n`.
- `NAME` – can be empty.

CheckpointsInfoIterator:: next()

This section describes the `CheckpointsInfoIterator` method.

The methods for the `CheckpointsInfoIterator` are the following:

```
CheckpointsInfoIterator::
    CheckpointInfo    next()
    CheckpointInfo[]  nextBatch(int count)
    void              close()
```

Where:

- The `next` method returns the next `CheckpointInfo` of the iteration, or `null` if the end of the iteration has been reached.
- The `nextBatch` method returns the maximum number of `CheckpointInfo` allowed for the iteration, or `less` if the end of the iteration has been reached.
- The `close` method is used to close the iteration. The `close` method must be called to release resources dedicated to the iteration within the Indexing System and inside the Helper.

The command uses the standard HTTP responses. See HTTP command response.

SyncedEntriesIterator::

The methods for the `SyncedEntriesIterator` are the following:

```
SyncedEntriesIterator::
   SyncedEntry   next()
   SyncedEntry[] nextBatch(int count)
   void           close()
```

- The `next` method returns the next document of the iteration, or `null` if the end of the iteration has been reached.
- The `nextBatch` method returns the maximum number of documents allowed of the iteration, or `less` if the end of the iteration has been reached.
- The `close` method is used to close the iteration. The `close` method must be called to release resources dedicated to the iteration within the Indexing System and inside the Helper.

The `SyncedEntry` object contains:

Data types

Member	Description
`uri`	A URI is an opaque string that uniquely identifies the document from the connector point of view.

Member	Description
	See also URI.
`stamp`	See Stamps.
`isFolder`	A boolean that is `true` if the entry refers to a directory, `false` otherwise.

```
class SyncedEntry
{
  String   getUri()
  String   getStamp()
  bool    isFolder()
}
```

SyncedEntriesIterator enumerateSyncedEntries(String rootPath, EnumerationMode enumerationMode)

Opens an iterator on a document and/or folder collection matching the `rootPath` given as parameter. It enumerates entries that have been pushed and are in `synced` status. It returns a stream of entries. An entry is made of a URI and a stamp.

The underlying idea of this method is to:

- Enumerate entries in the index.
- Decode the URI to find items in the data source.
- Test whether items still exist. If all items have been removed from the datasource, then:
  - delete the document,
  - or decode the stamp and check whether the items have been modified in the datasource.

Iterated results are streamed and used when needed.

Data types

A `SyncedEntriesIterator` is an abstract object which can be used to retrieve document statuses.

The object contains:

Types/flag	Description
`rootPath`	A part of the URI used to select a subset of the corpus. See also URI.
`enumerationMode`	The `EnumerationMode` lists the available types. For example, `NOT_RECURSIVE_ALL` returns the subfolders and the documents in the `rootPath`.

Types/flag	Description
	Similarly, `RECURSIVE_DOCUMENTS` returns all the documents in the `rootPath` (but not the subfolders).

```
enum EnumerationMode
{
 NOT_RECURSIVE_FOLDERS,
 NOT_RECURSIVE_DOCUMENTS,
 NOT_RECURSIVE_ALL,
 RECURSIVE_DOCUMENTS
}
```

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/enumerate_synced_
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
`PAPI_rootPath`	[URL]	The `rootPath` parameter is the string representation of the `rootPath`. It can take the form: `/root/subdir1/subdir2/subdir3/subdir3/...`
`PAPI_mode`	[URL]	The `mode` parameter is the string representation of the mode: • `NOT_RECURSIVE_FOLDERS` • `NOT_RECURSIVE_DOCUMENTS` • `NOT_RECURSIVE_ALL` • `RECURSIVE_DOCUMENTS`
`PAPI_session`	[URL]	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

Here is the response format for each entry:

```
[D/F] [space] [url_encode(URI)]          [space] [escape(STAMP)] [\n]
```

Where

- `url_encode()` – is a function which performs an url encoding of the given value.
- `escape()` – is a function which replaces `\r` and `\n` with `\\r` and `\\n`.
- `D/F` – `D` indicates an existing document, `F` indicates a folder.

Use of iterators with concurrent add and delete operations

- Add/Delete operations do not impact iterators that are already opened.
- Added/Deleted documents may not appear immediately in the iterated entries because of asynchronous treatment.

ulong countSyncedEntries(String rootPath, EnumerationMode enumerationMode)

Opens an iterator on a document and/or folder collection matching the `rootPath` given as a parameter, but only returns the number of items found.

Therefore, it counts the number of entries in the whole or in a subset of the Indexing corpus for that Connector.

Data types

The object contains:

Types	Description
`rootPath`	For details, see void deleteDocumentsRootPath(String rootPath [, Boolean recursive=true]).
`enumerationMode`	For details, see SyncedEntriesIterator enumerateSyncedEntries(String rootPath, EnumerationMode enumerationMode).

HTTP method

The method used is:

```
GET no-cache http://<host>:<port>/papi/4/connectors/<connectorName>/count_synced_entr
```

HTTP parameters

The parameters are described in the table below.

Parameter	Location	Description
PAPI_rootPath	[URL]	The rootPath parameter is the string representation of the rootPath. It can take the form: /root/subdir1/subdir2/subdir3/subdir3/...
PAPI_mode	[URL]	The mode parameter is the string representation of the mode: • NOT_RECURSIVE_FOLDERS • NOT_RECURSIVE_DOCUMENTS • NOT_RECURSIVE_ALL • RECURSIVE_DOCUMENTS
PAPI_session	[URL]	The optional parameter that retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

No Exception or error message is returned if the rootPath refers to an empty subset of the corpus. If status is OK, the body contains the string representation of the integer value.

Use of iterators with concurrent add and delete operations

- Add/Delete operations do not impact iterators that are already opened.
- Added/Deleted documents may not appear immediately in the iterated entries because of asynchronous treatment.

void sync()

The sync method can be used to flush all previous operations to disk since the last sync operation, to guarantee crash-proofness. It is a synchronous call that may take some time before returning control.

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
PAPI_session	[URL]	The optional parameter that retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/sync
```

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

void triggerIndexingJob()

The triggerIndexingJob method can be used to trigger the indexing job.

Important: In V6R2014 and higher versions, the triggerIndexingJob() method may commit an indexing job if a document analysis has been started. Unlike, the sync() method, this method does not block the PAPI.

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
PAPI_session	[URL]	The optional parameter that retrieves the session given by a previous call to get_current_session_id Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/trigger_indexing_job
```

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

boolean areDocumentsSearchable(long serial)

The `areDocumentsSearchable` method determines whether the documents can be found at search time. Use it with the `sync` method which provides the expected serial.

Note: The `setCheckpoint` method with the `sync` parameter set to `true` also provides the expected serial.

HTTP method

The method used is:

```
POST http://<host>:<port>/papi/4/connectors/<connectorName>/are_documents_searchable
```

HTTP parameter

The parameter is described in the table below.

Parameter	Location	Description
`PAPI_serial`	`[URL]`	The `serial` parameter is the string representation of the serial.
`PAPI_session`	`[URL]`	The optional parameter that retrieves the session given by a previous call to `get_current_session_id` Action: if there is a session mismatch, the Push API server refuses the command and returns an exception.

HTTP response

The command uses the standard HTTP responses. See HTTP command response.

If status is `OK`, the body contains the string representation of the boolean value (`true` or `false`).

Metadata Examples

The following table contains the Metadata name-value pairs that should be understood by the `addDocument` method.

Name	Format	Description	Example
`lastmodified`	RFC 822 and RFC 2822 formats ("RFC Date Format"), that is the common format in most	The date to be associated with the document.	1977/07/18-11:50:36 (GMT)1980/09/14

Name	Format	Description	Example
	Internet protocols (Mail, HTTP, ..) ISO 8601 and RFC 3339 formats Unix date and time (English format)		
`publicUrl`	URL	The public URL of the resource.	http://server /getDoc.php?id=24
`author`	Displayed string	Author name	John Doe
`mail:from`	See RFC 822	The sender of the document.	John Doe <doe@doe.net>
`mail:to`	See RFC 822		
`mail:cc`	See RFC 822		
`mail:bcc`	See RFC 822		
`language`	ISO 639	Primary or secondary level language tag	fr-FR en ar-AR
`security`	[~] PROVIDER:TOKEN Known providers: windows notes unix **Note:** The prefix ~ can be used for specifying a negative security token	You must add the special security token declaring that the document is public: `com.exalead.papi.hel`	windows:S-1-5-21-3495842611-1063732614-555398628-5176 or ~windows:S-1-5-21-3495842611-1063732614-555398628-5176 notes:cn=Doe/cn=Exalead/cn=com or ~notes:cn=Doe/cn=Exalead/cn=com
`file_name`	String	Name of the file.	
`file_size`	ulong	The size in bytes of the data associated to the document.	42

Name	Format	Description	Example
`title`	String	The title associated to the document.	

To create categories in the Exalead CloudView, the Indexing System considers both the original metadata and the metadata extracted from the document content. The priority rules for metadata may be configured in the Indexing System administration interface. For example:

- The Indexing System uses both the `mimeHint` and `filename` of the document master `Part`, and the content type detected by an analysis of the source to generate a `Top/Attributes/Kind` category.
- The Indexing System uses both the language meta and the detected language from the document text to generate a `Top/Attributes/Language` category.

Using the Push API Client

The Push API Client allows you to write applications in Java or C# that push documents to the index for all versions of Exalead CloudView

It is designed for Exalead partners and contractors who want to index new data sources for Exalead CloudView.

Note: See the Push API sample application delivered with the Exalead CloudView Connectors Java SDK or in the `/INSTALLDIR/sdk/cloudview-sdk-java-connectors/samples` directory.

Installing the Push API Client

Java project requirements

This section explains how to use the Push API V4 in your Java project:

Prerequisites	Files required	Recommended
Java v1.5 or later required	`papi-java-client.jar`	Java documentation for the Client library

.NET project requirements

This section explains how to use the Push API V4 Client in your .NET project.

Prerequisites	Files required	Recommended
Visual Studio 2005/2008 and higher .Net - version 2.0 and higher	`Exalead.PushApi.Clien` `Exalead.PushApi.Clien`	.NET documentation for the Client library

Instantiating the Push API Client

The first step in writing an application is to create a Push API client instance to connect to Exalead CloudView.

You must specify the connector name (in `connectorName`) that will be displayed in the Administration Console, for example, `myPapiApp`.

The connector name allows you to distinguish between two documents coming from two different connectors. Thus a document identifier in Exalead CloudView is made as follows: (`connectorName`, `documentURI`).

Example 1. Java code

```
import com.exalead.papi.helper.*;
final PushAPI papi = PushAPIFactory.createHttp(PushAPIVersion.PAPI_V4, host, port, co
connectorType, login, password);
```

Example 2. C# code

```
using Exalead.PushApi.Client;
PushAPI papi = PushAPIFactory.CreateHttp(host, port, connectorName, connectorType, lo
```

Operations and states

Operations

The basic PAPI operations are the following:

Operation	Description
`AddDocument`	Creates a document including the parts, URI, metas, etc.
`GetDocumentStatus`	Using the documents URIs, it retrieves the stamps and their statuses to determine the statuses of documents.
`Delete`	The following methods are available: • delete a document identified by its URI.

Operation	Description
	• `deleteDocumentRootPath` – deletes a collection of documents (recursively or not) matching a given root path. The hierarchy is based on the slashes "`/`"of the URI. • `deleteDocumentsWithPrefix` – deletes all child documents under a root prefix. This method is useful if the document hierarchy is not based on slashes, but can be risky as it does not have any filtering mechanism. For example, if you have the following documents: ◦ `eno:bo:master` ◦ `eno:bo:masterpart` ◦ `eno:bo:masterpart:subpart1` ◦ `eno:bo:master:subpart2` Using `deleteDocumentsWithPrefix("eno:bo:master")` will delete all documents with this prefix, even if you want to delete `eno:bo:master` and `eno:bo:master:subpart2` only.
`Enumerate`	Lists the Exalead CloudView documents and their stamps, given a URI prefix (recursively or not).
`Checkpoints`	Sets, retrieves or removes checkpoints. For example, to perform incremental updates easily.
`UpdateDocument`	Updates a document including parts and metas.

Document statuses

Because some operations are performed asynchronously (add, delete, set checkpoint), it's important to know at which state of the processing flow the command is performed.

The above operations can have the following document statuses:

Status	Description
`Initial`	Data is only in the source (not already sent to the index by the connector).
`Pushed`	The document has been received by the Exalead CloudView index, but can be lost if a crash occurs before the sync. All `Add()`, `Delete()` operations create a document in the `Pushed` state. `GetDocumentStatus()`, `GetCheckpoint()`, `EnumerateCheckpointsInfo()` operate on documents pushed, synced, and searchable.

Status	Description
`Synced`	The document has been securely synced to disk. This guarantees crash-proofness. Documents cannot be lost. `EnumeratesSyncedEntries()` operates on document synced.
`Searchable`	The document can be found at search time.

Session handling

The Push API session handling mechanism helps guarantee the consistency of a stream of PAPI operations, even in the event of a PAPI server restart between operations.

The mechanism can be used by calling `startPushSession()` before sending your PAPI operations, and `stopPushSession()` at the end of your operations. If an unexpected restart occurs, the next operation will trigger a session identifier mismatch.

Internally, `startPushSession` asks the remote Push API server for its current session identifier, created when the server starts. This identifier will be used in all further client commands to ensure that the server keeps the same identifier. Any command with a session identifier mismatch will fail with an error. `stopPushSession` clears the current session identifier. All further client commands will no longer use the session identifier after this call.

Note: For a managed connector, the `startPushSession()` and `stopPushSession` calls are made automatically by the framework and do not have to be included in the connector code. For an unmanaged connector, the `startPushSession()` and `stopPushSession` calls must be included manually in the connector code.

Indexing your first PAPI document

Run the sample program

This section explains how to run the sample program to index your first document.

Example 3. Java Code

```
import com.exalead.papi.helper.Document;
import com.exalead.papi.helper.Meta;
import com.exalead.papi.helper.Part;
// [...]
final PushAPI papi = createConnection(...);
//new document (uri , stamp)
final Document doc = new Document("doc1", "2014-03-15");
// create the metas
```

```
doc.addMeta(new Meta("title", "My document's title"));
doc.addMeta(new Meta("date", "2014-03-20"));
doc.addMeta(new Meta("size", "5493"));
doc.addMeta(new Meta("approved", "false"));
// master part
final byte[] bytes = new String("the text to index...").getBytes("UTF-8");
// if you don't specify part name, the part is considered as Master part
final Part masterPart = new Part(bytes);
masterPart.setEncoding("UTF-8");
masterPart.setFileName("filename.txt")
doc.addPart(masterPart);
// another part
final Part part = new Part("Second part",bytes);
part.setEncoding("UTF-8");
part.setExtension("txt");
doc.addPart(part);
// push the document
papi.addDocument(doc);
```

Example 4. C# Code

This code snippet demonstrates how to send the document.

```
//How to send a document.
void IndexDocument()
{
    Document doc = new Document("doc1");
    // the stamp associated to the document
    doc.Stamp = "2014-03-15";
    // create the metas
    MetaContainer metaContainer = new MetaContainer();
    metaContainer.AddMeta(new Meta("title", "My document's title"));
    metaContainer.AddMeta(new Meta("date", "2014-03-20"));
    metaContainer.AddMeta(new Meta("size", "5493"));
    metaContainer.AddMeta(new Meta("approved", "false"));
    doc.MetaContainer = metaContainer;
    PartContainer partContainer = new PartContainer();
    // master part
    byte[] bytes = new UTF8Encoding().GetBytes("the text to index...");
    Part masterPart = new Part(bytes);
    masterPart.Encoding = "UTF-8";
    masterPart.Filename = "foo.txt";
    partContainer.AddPart(masterPart);
    Part part = new Part(bytes);
    part.Encoding = "UTF-8";
    part.Filename = "foo.txt";
    partContainer.AddPart(part);
    doc.PartContainer = partContainer;
    // push the document
```

```
    papi.AddDocument(doc);
}
```

How to force the indexing of pending operations

To force indexing, you must call the two following methods.

Example 5. Java Code

```
// This forces a flush to disk
papi.sync()
// This triggers the indexing of committed documents.
// In V6R2014 and higher, the task queue is optional (no task queue by default)
// If there is no task queue, the following method may commit an indexing job if a
// document analysis has been started. Unlike, the sync method, this method does not
// block the PAPI
papi.triggerIndexingJob()
```

Example 6. C# Code

```
// This forces a flush to disk
papi.Sync()
// This triggers the indexing of committed documents.
// In V6R2014 and higher, the task queue is optional (no task queue by default)
// If there is no task queue, the following method may commit an indexing job if a
// document analysis has been started. Unlike, the sync method, this method does not
// block the PAPI
papi.TriggerIndexingJob()
```

Important: In V6R2014 and higher versions, the `triggerIndexingJob()` method may commit an indexing job if a document analysis has been started. Unlike, the `sync()` method, this method does not block the PAPI.

Important: In Exalead CloudView V6, the `sync()` method should not be called by the connector during standard indexing. It is controlled by the **Force Indexing after scan** option in the Administration Console **> Connectors > Deployment > Push API** section. When this option is selected, Exalead CloudView will automatically trigger the indexing job after each scan. You should use the `sync()` method for very specific use cases only. For example, if you need to make a diff between indexed documents in Exalead CloudView and documents in the source. In that case, you must: push new documents, make a `sync()` to trigger the indexing job, then enumerate synced entries to make a diff with your source.

Check the document status

You can use `GetDocumentStatus` to retrieve the status of a specific document using its URI.

Example 7. Java Code

```
void getDocumentStatus() throws PushAPIException {
    final String uri = "doc1";
    final DocumentStatus ds = papi.getDocumentStatus(uri);
    if (ds.isExist()) {
      System.out.println("EXISTS! Stamp = " + ds.getStamp());
    } else {
      System.out.println("MISSING!!!");
    }
  }
```

Example 8. C# Code

```
public void GetDocumentStatus()
{
    string uri = "doc1";
    DocumentStatus ds = papi.GetDocumentStatus(uri);
    if (ds.Exist)
        Console.WriteLine("EXISTS! Stamp = " + (ds.Stamp ?? "(null)"));
    else
        Console.WriteLine("MISSING!!!");
}
```

Indexing a Document Collection

This section explains how to set the document URI to the filesystem path, so that the URI tree structure reflects the filesystem tree structure (required to be able to use PAPI enumeration to detect new and deleted documents).

The following code snippets demonstrate the new Document constructor.

Example 9. Java Code

```
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import org.apache.commons.io.IOUtils;
import org.apache.log4j.Logger;
import com.exalead.papi.helper.Document;
import com.exalead.papi.helper.Part;
import com.exalead.papi.helper.PushAPI;
import com.exalead.papi.helper.PushAPIException;
public class FolderIndexer {
    public FolderIndexer(final PushAPI papi, final Logger logger) {
        this.papi = papi;
        this.logger = logger;
    }
```

```
    void index(final File folder) {
        for (final File file : folder.listFiles()) {
            if (file.isFile()) {
                try {
                    final InputStream stream = new BufferedInputStream
                        (new FileInputStream(file));
                    try {
                        final byte[] bytes = IOUtils.toByteArray(stream);
                        final Document doc = new Document(file.getAbsolutePath(),
String.valueOf(file.lastModified()));
                        doc.addPart(new Part(bytes));
                        papi.addDocument(doc);
                    } catch (final IOException e) {
                        logger.error("Could not read file " + file.getAbsolutePath(),
                    } catch (final PushAPIException e) {
                        logger.error("Could not send file to indexing server", e);
                } catch (final FileNotFoundException e) {
                    logger.error("File does not exist: " + file.getAbsolutePath(), e)
                }
            }
        }
    }
    private final PushAPI papi;
    private final Logger logger;
}
```

Example 10. C# Code

```
public void IndexDocumentCollection()
        {
            foreach (string uri in Directory.GetFiles("."))
            {
                Console.WriteLine("Push document : " + uri);
                Document doc = new Document(uri);
                FileInfo fileInfo = new FileInfo(uri);

                // the stamp associated to the document
                doc.Stamp = fileInfo.LastWriteTime.ToString();

                // create the metas
                MetaContainer metaContainer = new MetaContainer();
                metaContainer.AddMeta(new Meta("creation_date",
                     fileInfo.CreationTime.ToString()));
                metaContainer.AddMeta(new Meta("size",
                     fileInfo.Length.ToString()));
                doc.MetaContainer = metaContainer;
                PartContainer partContainer = new PartContainer();
                // master part
                byte[] bytes = File.ReadAllBytes(uri);
```

```
                Part masterPart = new Part(bytes);
                masterPart.Extension = fileInfo.Extension;
                partContainer.AddPart(masterPart);
                doc.PartContainer = partContainer;

                // push the document
                papi.AddDocument(doc);
            }
        }
```

Listing Synced Documents

Before listing the Exalead CloudView index, you need to ensure that changes sent by the connector are taken into account. You can do so by using the `sync` method of the Push API, or by calling the `setCheckpoint` method with the `sync` parameter set to "`true`".

Checkpoints

This section explains what is a checkpoint and how to use it.

A checkpoint is a string value associated to a connector. Multiple checkpoints are possible for a connector.

Two methods are used to manipulate the checkpoint: `Get` and `Set` methods.

Note: `Get` operates on `Synced` state => a `Get` performed immediately after a `Set` may not return the same value.

The `setCheckpoint` method can be used to sync the previous operations by setting the `sync` parameter to `true`. It can also be used to know whether the operation status is `Searchable`. See Operations and states.

Typical use cases of checkpoint operations:

- Store the last synchronization date of a folder.
- Store the last eventId in a journal of events.
- Safely sync to disk previous operations (`sync` set to `true`).
- Allow tracking of operations state to know whether documents are searchable; see Operations and states.

Sync code snippet

Below is a code sample to sync documents at the end of the scan operation.

Example 11. Java Code

```
// Example of sync.
public void SyncDocuments(final Date indexingStartDate) {
papi.setCheckpoint(String.valueOf(indexingStartDate), "sync date", true);
}
```

Example 12. C# Code

```
// Example of sync.
        public void SyncDocuments(DateTime indexingStartDate)
{
    papi.SetCheckpoint(indexingStartDate.ToString(), "sync date", true);
}
```

List documents

Below is a code sample to enumerate the documents of a specified folder.

Example 13. Java Code

```
for (final SyncedEntry doc : papi.enumerateSyncedEntries("myfolder", EnumerationMode.
  System.out.println("uri : " + doc.getUri() + " stamp : " + doc.getStamp());
}
// no need to close Iterator because the end is reached
```

Example 14. C# Code

```
//Example of the enumeration of document in a specified folder.
public void Enumerate()
{
    string rootPath = "myfolder";
    EnumerationMode mode = EnumerationMode.RECURSIVE_DOCUMENTS;
    foreach (SyncedEntry entry in new
    SyncedEntriesEnumerator(papi.EnumerateSyncedEntries(rootPath, mode)))
    {
        Console.WriteLine("URI = " + entry.Uri + ", Stamp = " + (entry.Stamp ?? "(nul
    }
}
```

Updating Documents

This section explains which methods you should use for partial and full document updates.

- For partial updates, use the `updateDocument` methods. See void updateDocument(Document document, string[] fields) and void updateDocumentList(Document[ ] documentList, string[][] fieldsList).

- For full updates, use the `addDocument` method. See void addDocument(Document document) and void addDocumentList(Document[ ] documentList).

Monitoring the Index

You can use the `areDocumentSearchable` method to know whether the last pushed documents are searchable.

Example 15. Java Code

```
static void waitForDocumentsAreSearchable() throws PushAPIException {
    // get the serial that can be used to track processing status of documents
    final BigInteger serial = papi.setCheckpoint("no value", "foo");
    // trigger indexing job
    papi.triggerIndexingJob();
    // wait for documents searchable
    try {
      while (!papi.areDocumentsSearchable(serial)) {
        Thread.sleep(1000);
      }
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
  }
```

Example 16. C# Code

```
static void WaitForDocumentsAreSearchable()
{
    // get the serial that can be used to track processing status of documents
    ulong serial = papi.SetCheckpoint("no value", "foo");
    // trigger indexing job
    papi.TriggerIndexingJob();
    // wait for documents searchable
    while (!papi.AreDocumentsSearchable(serial)) {
        Thread.Sleep(1000);
    }
 }
```

Push API Connector Framework

The Push API Connector Framework allows you to develop custom Java connectors that push documents to Exalead CloudView

This connector can be deployed, configured and managed using the Exalead CloudView functions.

The Framework is designed for Exalead partners and contractors who want to index new data sources in Exalead CloudView.

The development framework delivered with Exalead CloudView is located in `<INSTALLDIR>\sdk\`

In this package, you will find all the components that you need to build connectors or security sources, that can be plugged and managed by Exalead CloudView using the Java programming language.

The SDK contains the following directory and files:

- `sdk/java-customcode/docs/api` – API Documentation
- `sdk/java-customcode/lib/`
  - `papi-java-client.jar`
  - `papi-java-connector.jar`
  - `security-java-api.jar`
- `sdk/java-customcode/samples/connectors/samples/`
  - `example-filesystem-connector.jar`

Connector Framework Prerequisites

To develop a custom connector using the Push API Java Connector Framework, you must first create a Java project, for example, in Eclipse. The requirements and dependencies are detailed below.

Note: We recommend using the Exalead CloudView Eclipse plugin that allows you to easily develop, package and deploy your custom components.

Global Requirements

Global requirements for JAR projects are:

- CloudView V6.x
- a Java JDK (version 1.5 or later)

Dependencies

A Push API Connector is a JAR project that has the following dependencies:

- `papi-java-connector.jar` (contains `Connector` definition)
- `papi-java-client.jar` (contains `PushAPI` definition)

All dependencies should be used to compile your project.

Using the Eclipse plugin

The Exalead CloudView Eclipse plugin is provided to help you develop and deploy plugins in Eclipse Indigo 3.7 or later. The documentation is packaged with this plugin and is available at `http://eclipse.exalead.com`.

You can develop custom components for:

- Exalead CloudView core
  - Connectors
  - Document Processors
  - Meta Processors
  - Prefix Handlers
  - Push API Filters
  - Query Processors
  - Security Sources
- Exalead CloudView Mashup
  - Widgets
  - Feeds

- Feed Triggers
- Mashup Triggers
- Pre-Request Triggers
- MEL Functions
- Security Providers

Example 17. Why use it

- Easy deployment: you can package your plugin with customized components to be exported and deployed automatically on the selected instance of Exalead CloudView. For Mashup UI, this avoids rebuilding and redeploying the `standalone-mashup-ui.war` package for each customized item.
- Quick export: you can package your plugin with classes that you want to export and then export it as a zip file on a selected path.
- Manage installs: you can list the deployed plugins on a selected instance of Exalead CloudView and then select the plugins to remove.
- Debug connectors: you can use the debugging functionality to debug the code of custom connectors on-the-fly and avoid redeploying on an instance to check your code changes.

Implementing the Connector

This section describes how to implement your connector.

Manage the configuration

To manage the connector's configuration, you must first extend the `ConnectorConfig` class. Your class must follow the recommendations described in Top level configuration class(es).

A basic filesystem connector needs to know where to start crawling for files. By specifying a root folder, the connector will recursively crawl all the files located in this folder.

See the `DemoFileSystemConnectorConfig` sample code:

```
package com.exalead.papi.framework.connectors.example.filesystem;
import com.exalead.papi.framework.connectors.ConnectorConfig;
import com.exalead.config.bean.IsMandatory;
import com.exalead.config.bean.PropertyLabel;
import java.io.File;
/**
  * The configuration class for the filesystem connector.
  */
public class DemoFileSystemConnectorConfig extends ConnectorConfig {
```

```
     private File startingFolder = null;
     @IsMandatory(false)
     @PropertyLabel("My starting folder")
     public void setStartingFolder(final File startingFolder) {
         this.startingFolder = startingFolder;
     }
     public File getStartingFolder() {
         return startingFolder;
     }
     /**
      * Optional method listing all option names as they will be displayed in the UI
      **/
     public static String[] getMethods() {
         return new String[] { "StartingFolder" };
     }
}
```

Encrypt the password

If your connector requires an encrypted password to connect to the source (for example, a database), then you must define a `Password` property in the configuration. You add a setter method as follows.

```
...import com.exalead.config.security.Crypter;
public class MyDataBaseConf extends ConnectorConfig{
    private String password;
   // setter to define Password property
    @BeeKeyValueType("encrypted")
    public void setPassword(final String password) {
      this.password = Crypter.getInstance().decrypt(password);
    }
...
```

Implement the connector

To implement a Java connector, you must extend the `Connector` class. The constructor of your class must have your ConnectorConfig-derived class as sole parameter.

The second class is your `ConnectorConfig`, in this case, `BasicFilesystemConnectorConfig`

```
public class BasicFilesystemConnector extends Connector {
    private final BasicFilesystemConnectorConfig config;
    public BasicFilesystemConnector(final BasicFilesystemConnectorConfig config)
         throws Exception {
        super(config);
        this.config = config;
    }
```

Note: The previous version of the constructor, taking an additional PushAPI object as first parameter, is now deprecated and must not be used.

The following methods must be implemented:

Method	Description
`public void scan(PushAPI papi, String mode, Object modeConfig) throws Exception;`	This method is used to scan/synchronize all documents. The provided PushAPI object is to be used for synchronization operations. The optional mode parameter, and its optional configuration object, can be used in specialized scan cases defined with the `@ConnectorCapabilities` annotation. You can also set the `@ConnectorCapabilities` annotation to get a continuous scan. See Implement a continuous scan.
`public Document fetch(String uri) throws Exception`	This method is used to retrieve a document from the source, for example, to create a thumbnail or when a user clicks on a search result. `uri` is the URI of the indexed document.
`public MetaContainer getDocumentSecurityTokens(String uri);`	This method is used to retrieve the security tokens of a document. This method is not used during indexing but when Exalead CloudView needs to retrieve security tokens at real time. `uri` is the URI of the indexed document. When you push documents, you must add a security meta: `document.addMeta("security", "mytoken");`

You can use the following helper to self-abort a synchronization in progress:

```
/**
     * Helper which can be called by the connector code to self-abort.
     * @param reason the reason why the abort was issued ; may be null
*/
     @Override
     public final void selfAbortScan(String reason);
```

You can also use the following triggers, which are called upon aborted scan, suspended scan, or resumed scan, to execute additional operations:

```
/**
            * Called by the framework when scan abort is requested.
            * This method must not change the state of the connector, only wake up sc
            * which might be blocking.
            * The scan will only be considered as aborted when the scan() method has
        */
     @Override
     public void onScanAborted() throws Exception;
/**
            * Called by the framework when scan suspend is requested.
            * This method must not change the state of the connector, only wake up sc
            * which might be blocking.
            * It is then the responsibility of the connector to set the status of the
            * to SUSPENDED when effectively taken into account.
        */
     @Override
     public void onScanSuspended() throws Exception;
/**
            * Called by the framework when scan resume is requested.
            * This method must not change the state of the connector, only wake up sc
            * which might be blocking.
            * It is then the responsibility of the connector to set the status of the
            * to WORKING when effectively taken into account.
        */
     @Override
     public void onScanResumed() throws Exception;
```

You should also call the following helper regularly inside long worker loops (long enumeration of documents) to be able to exit gracefully if an abort has been requested by the user, or by the internal framework:

```
/**
            * Checks the current connector status and, if an abort command was
            * sent, throws an ConnectorAbortingException exception.
            *
            * This function is a helper which can be called by connector. It is not
            * called by the framework.
        **/
     @Override
     public void checkAbortingOperation() throws ConnectorAbortingException;
```

See the sample Java code for a basic filesystem connector (`DemoFileSystemConnector.java`) located in `<INSTALLDIR>\sdk\cloudview-sdk-java-connectors\samples\fsbasic`.

Implement a continuous scan

Scan modes describe how connectors index documents. By default, a connector has one scan mode called `full` which starts when you click the **Scan** button in the Administration Console. Additional scan modes can be developed and provided with the connector. They are started when you select **CONNECTOR NAME > Operation > More actions** and click their corresponding **Run** buttons.

Scan modes are described by a `ScanModeDefinition`. A `ScanModeDefinition` contains a workflow which describes how a connector will process the indexing. The workflow can be:

- a scan-based indexing (`Wokflow.SCAN_BASED`)
- or a permanent scan (`Workflow.PERMANENT_WORK`).

A scan-based indexing exits when the scan is done. The connector starts, scans and indexes documents in the `scan()` method, and quits. This method is either called periodically to index new documents automatically, or manually by clicking the **Scan** button.

A permanent-work indexing (also called continuous scan) does not exit when a first scan is done. The connector loops forever in the `scan()` method, indexing new documents permanently. This method is started automatically by Exalead CloudView just after the connector's initialization, so that when Exalead CloudView starts, your connector starts running immediately. When a connector is set to permanent-work mode, we recommend implementing an "abort" command, to let users click the **Abort scan** button, when they want the connector to terminate its job and exit.

The following code sample show how to implement the continuous scan.

```
@ConnectorCapabilities(
    scanModes = {
        @ScanModeDefinition(
            name = "full",
            workflow = ConnectorCapabilities.Workflow.PERMANENT_WORK)
    }
)public class ContinuousScanConnector extends Connector implements CVComponent {
    public ContinuousScanConnector(ContinuousScanConfig config) throws Exception {
        super(config);
         }
         @Override
    public void scan(final PushAPI papi, final String scanMode, final Object scanMode
            throws Exception {
                  while( true ) {
            try {
                // do the job, index documents
                // ...
                // check for an abort
                if( getStatus(scanMode) == ConnectorStatus.ABORTING ) {
```

```
                    // log the abort and quit
                    break;
                }
            }
            catch(final Exception e) {
                // log the exception
                // handle the problem
            }
            finally {
                // clean everything
                // be ready for a next run
            }
        }
    }
}
```

Implement concurrent scan modes

By default, the connector framework is set to launch scans one after the other. You cannot run several scan operations at the same time mainly for thread safety.

For example, let's say that your connector is scheduled to launch full scan operations on a regular basis (for example, every 5 minutes) and that you want to run another scan operation once a week to update the index regarding what was deleted in the source content. The two scan operations may conflict at a given time and the second operation may not even be triggered.

To tackle this issue, you can set your connector behavior to allow concurrent scan modes using the `ConnectorCapabilities#isReentrant()` annotation property set to `false`.

Validate the connector configuration

You can write a config check class to validate the configuration of your connector. This class must implement the `CVComponentConfigCheck` interface and override the `check()` method. In this method, you should check whether all parameters of your configuration contain valid values.

If a parameter contains an invalid value, you should throw a `ConfigurationException` with a message describing the problem. The `check()` method is either called when you click the **Check config** button in the connector **Configuration** tab, or when you click the **Apply** button. Throwing a `ConfigurationException` will stop the validation process and you can't apply an invalid configuration.

The check method may also be called by `buildgct` when Exalead CloudView is not started.

Let's suppose that your connector is in the `MyConnector` class and its configuration in the `MyConnectorConfig` class.

```
public class MyConnectorConfigCheck implements CVComponentConfigCheck<MyConnectorConf
```

```
 @Override
 public void check(final MyConnectorConfig config, final boolean useNow) throws Confi
Exception {
 // use the MyParam getter
 final int myParam = config.getMyParam();
 // let's check the parameter value
 if( myParam < 0) {
 final ConfigurationException e = new ConfigurationException("Invalid MyParam value (
 e.setConfigKey("MyParam"); throw e;
  }
 }
}
```

And you should annotate your connector with a `@CVComponentConfigClass`.

```
@CVComponentConfigClass(configClass = MyConnectorConfig.class,
    configCheckClass = MyConnectorConfigCheck.class)public class MyConnector extends
```

Add logging capabilities

When a connector runs many things can happen. Knowing which exceptions where met by your connector is necessary to fix issues. This is why it is better to avoid creating your own loggers using either the standard java logger or the Log4J `Logger.getLogger()`.

We recommend using the `Connector.getLogger()` method, or even better, the `Connector.getLogger(String suffix)` method. These methods create loggers with the connector instance name. This allows you to differentiate multiple instances of the same connector. Use the method with the suffix argument when you have multiple classes.

```
public class MyConnectorClass extends Connector implements CVComponent {
    Logger thisClassLogger = getLogger();
    MyConnectorSubClass sub = new MyConnectorSubClass(getLogger("MyConnectorSubClass"
    // ...
}
public class MyConnectorSubClass {
     public MyConnectorSubClass(Logger logger) {
          // ...
     }
}
```

When performing tasks within other tasks, you can use `com.exalead.log4ng.Log4NGContext` which will append pushed contexts before each logging message in the same order they were pushed. This context stack is different for each thread.

For example:

```
Log4NGContext.push("State1");
getLogger().error("FIRST LOGGER");
```

```
Log4NGContext.push("State2");
getLogger("myLogger").error("SecondLogger");
Log4NGContext.pop();
Thread t = new Thread(new Runnable() {
 @Override
 public void run() {
 Log4NGContext.push("Thread");
 getLogger("myLogger2").error("ThrirdLogger");
 }
});
t.start();
t.join();
getLogger().error("FIRST LOGGER AGAIN");
Log4NGContext.pop();
getLogger().error("FIRST LOGGER ONE LAST TIME");
```

Would print something like:

```
State1: FIRST LOGGER
State1: State2: SecondLogger
Thread: ThrirdLogger
State1: FIRST LOGGER AGAIN
FIRST LOGGER ONE LAST TIME
```

Update the connector status

To update the connector scan status, that is to say the number of documents pushed, deleted and scanned, you must call the following methods during the scan:

`getState("scan mode").incPushed();`

`getState("scan mode").incDeleted();`

`getState("scan mode").incScanned();`

Where `"scan mode"` is the name of the scan passed to the scan method.

Packaging the connector as a plugin

Plugins are components or resources that can be hot plugged without restarting the product. They are installed in the DATADIR, and are therefore instance-wide.

To load your custom code in Exalead CloudView, it must be packaged as a CVPlugin. You can then easily upload it in the Administration Console.

Plugins can include special component classes, such as connectors, security sources, format converters, analysis processors, etc. They can also include other materials, such as resources (linguistics, statistics, etc.).

Each component class within a plugin can have its own configuration. For example, a connector, will usually have user-defined settings, which will be edited through the Administration Console > **Connectors** menu. The framework will handle:

- the configuration editing process,
- the connector instantiation.

Plugin structure

The plugin/component framework allows you to package a set of components within a plugin. The plugin will typically be deployed from a standalone ZIP file including all the necessary JARs and resources.

What is a plugin

A plugin is a regular ZIP file using the `1989 PKZip` original format, with deflate compression or no compression (beware to produce compatible files), and must have the following structure:

- `META-INF/cvplugin.properties` - can define several core properties related to the plugin, as key=value lines terminated by a line feed.

  Note:

  Properties can only use ASCII characters, a restriction that Java annotations do not bear for properties such as "Author" or "Copyright". See Description of the META-INF/cvplugin.properties file.

- `lib/` - subdirectory contains the plugin JAR file(s). The JAR file(s) will be parsed by the framework to locate all classes, load the required ones, execute initialization steps, etc.
- Other sub-directories can be included to collect additional resources. We strongly recommend not to put any standalone file at the top level of the ZIP directory structure, and collect necessary material in a dedicated sub-folder, such as a `resources/` sub-folder.

Description of the META-INF/cvplugin.properties file

Component	Description
`plugin.jars`	A space-separated list of JAR names to be parsed. If defined, the framework will not scan all the jar files present in the `lib/` subdirectory, but only scan those defined in this list. Example: `plugin.jars=mycomponent.jar`
`plugin.mainClass`	The main plugin class name. If defined, the framework will use the given class name instead of searching for a unique CVPlugin-derived class. Example: `plugin.mainClass=com.example.plugins.Plugin`
`plugin.author`	The author name. We recommend using the `@CVPluginAuthor()` annotation instead.
`plugin.copyright`	The copyright information. We recommend using the `@CVPluginCopyright` annotation instead.
`plugin.description`	The description. We recommend using the `@CVPluginDescription` annotation instead.
`plugin.version`	The version information. We recommend using the `@CVPluginVersion` annotation instead.

Create a basic plugin

The following procedure describes the main steps to create a CVplugin on a Linux environment.

1. Go to your plugin directory:
2. Create the `META-INF/` and `lib/` subdirectories:
3. Create a `cvplugin.properties` file under the `META-INF/` subdirectory:
4. Copy your JAR component in the `lib/` subdirectory:
5. Zip your plugin:

About the CVPlugin public class

To execute specific initializations through the constructor, you should define a public main class extending the `CVPlugin (com.exalead.mercury.plugin.CVPlugin)` class.

CVPlugin class basics

This class must be included in one of the JAR files of the `lib/` subdirectory. All classes within the JAR collection are available by default through the plugin class loader.

If no such class exists, and if there is only one component within the plugin, the framework will use metadata associated with this unique component.

The name of the CVPlugin-derived class is not important. There should be at most one CVPlugin-derived class in the classes collection.

```
/** My plugin. **/
@CVPluginDescription("My Wonderful Plugin")
@CVPluginVersion("1.0")
@CVPluginCopyright("Copyright Wonders & Co., All Rights Reserved")
@CVPluginAuthor("Wonders & Co.")
public class Plugin extends CVPlugin {
     protected final File installDirectory;
     /** Default constructor. **/
     public Plugin(final String name, final File installDirectory) {
         super(name, installDirectory);
         this.installDirectory = installDirectory;
         // Optionally, define the plugin initialization code here:
     }
}
```

CVPlugin class annotations

This CVPlugin-derived plugin class may define a number of useful annotations (located in the `com.exalead.mercury.plugin` package) to provide additional information.

Annotation	Description
`@CVPluginAuthor`	The plugin author name(s).
`@CVPluginCopyright`	The plugin copyright information.
`@CVPluginDescriptio`	The plugin description.
`@CVPluginVersion`	The plugin version information.

Top level component class(es)

A component is a class implementing the `com.exalead.mercury.component.CVComponent` interface.

This top-level interface is an empty interface (no defined method inside) designed to be automatically detected by the framework.

All available non-inner classes implementing this interface are collected during startup while scanning all JARs, and allow to list a subset of components implementing a specialized class or interface at runtime. For example, all components deriving from the `com.exalead.papi.framework.connectors.Connector` class are listed to collect the list of available connectors in the product.

Top level component class annotations

Annotation	Description
`@CVComponentDescription` (`com.exalead.mercury.component.CVComponentDescription`)	If specified, this is the short description of the connector used in the select box of the **Add connector** dialog box in the Administration Console. Example: `@CVComponentDescription("My Wonderful Connector Component")`
`@CVComponentConfigClass` (`com.exalead.mercury.component.conf` `CVComponentConfigClass`)	This annotation defines the: The associated `CVComponentConfig` derived class used for the configuration. `CVComponentConfigCheck` (`com.exalead.mercury.component.config.CVCom` used to enhance the configuration check of the CVComponent. Example: `@CVComponentConfigClass(configClass = FilesystemConnectorConfig.class, configCheckClass = ConnectorConfigCheck.class)`
`@CVComponentLabel` (`com.exalead.mercury.component.CVC`	If specified, this is the label of the component. The label is used in the Administration Console when selecting a custom document processor, semantic processor, or query prefix handler.

Annotation	Description
	For example: `@CVComponentLabel("My document processor")`
`@PropertyLabel (com.exalead.config.bean.PropertyLa`	DEPRECATED Similar to the `@CVComponentDescription` annotation.
`@IsEmptyConfig (com.exalead.config.bean.)`	The given configuration class is empty (no setters at all). Without this annotation, an empty class would be rejected at configuration build time.
`@IntrospectableComponent (com.exalead.mercury.component. IntrospectableComponent)`	This annotation defines the: `CVComponentIntrospector (com.exalead.mercury.component.CVComponentI` derived co-class used for introspection queries, for example the **Check connectivity** operation available in the Administration Console. and a list of supported query classes derived from `SupportedQuery (com.exalead.mercury.component.SupportedQue`

Example:

```
@IntrospectableComponent(introspectorClass = MyConnectorIntrospector.class, supported
        @SupportedQuery(queryClass = CheckConnectivity.class),
 @SupportedQuery(queryClass = TestConnection.class),
        @SupportedQuery(queryClass = ListDirs.class),
 @SupportedQuery(queryClass = ListFiles.class),
        @SupportedQuery(queryClass = ListForms.class),
 @SupportedQuery(queryClass = ListItems.class),
        @SupportedQuery(queryClass = ListViews.class) })
```

Top level configuration class(es)

A configuration class defines a list of configuration properties that can be used by a component.

The framework will usually:

- manage the serialized configuration (as an XML object),
- unserialize it,
- create an instance of the given class, and allow you to edit its properties,
- handle the configuration of components from the Administration Console.,

- etc.

The exact workflow is specific to each component type. For example, some component types may not have any configuration at all.

A configuration class must implement the empty `CVComponentConfig (com.exalead.mercury.component.config.CVComponentConfig)` interface to be accepted as a valid configuration class.

Example:

```
/**
  * The configuration class for the filesystem connector.
**/
@CVComponentDescription("Filesystem simple demo (java)")
public class DemoFileSystemConnectorConfig extends ConnectorConfig
{
     private File startingFolder = null;
     @IsMandatory(false)
     public void setStartingFolder(final File startingFolder)
     {
         this.startingFolder = startingFolder;
     }
     public File getStartingFolder()
     {
         return startingFolder;
     }
     /**
       * Optional method listing all option names as they will be displayed in the UI
     **/
     public static String[] getMethods()
     {
         return new String[] { "StartingFolder" };
     }
}
```

Note: An additional static method named `getMethods` may be defined in a configuration class, to return properties in a specific order. This method should be public, taking no argument, and returning an array of String corresponding to the ordered property names.

Example :

```
public static String[] getMethods()
     {
         return new String[]{"Proxy", "ProxyPort"};
     }
```

Top level configuration additional interfaces

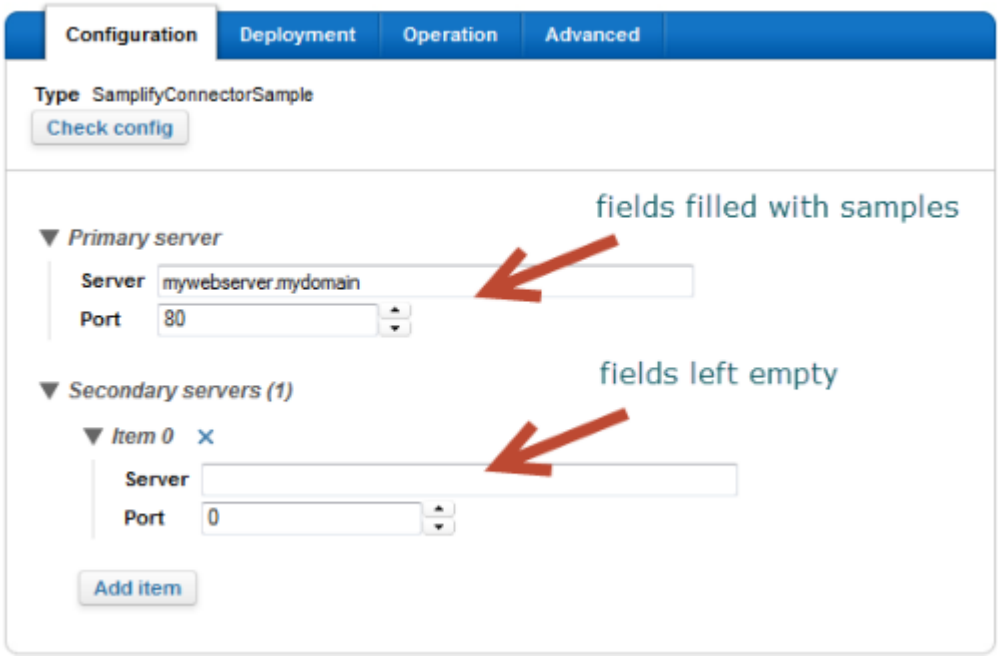
Interface	Description
`CallAfterFill (com.exalead.config.bean.CallAfterF`	Allows you to define a `callAfterFill()` observer method to be called upon filling. This method is used to perform specific post-actions related to filled properties. This interface is not required to get a working configuration.
`CVComponentConfigSamplify (com.exalead.mercury.component.conf CVComponentConfigSamplify)`	Allows you to define a `samplify()` method, that will be called when a sample object is requested by the framework. This method is used to fill properties with a real-world example. For example, a `Hostname` property may be filled with `myhostname`

The following example shows the `CVComponentConfig` class using the `samplify` method.

```
package com.exalead.connectors;
import com.exalead.papi.framework.connectors.ConnectorConfig;
import com.exalead.config.bean.ConfigurationException;
import com.exalead.config.bean.IsMandatory;
import com.exalead.mercury.component.config.CVComponentConfig;
import com.exalead.mercury.component.config.CVComponentConfigSamplify;
public class SamplifyConnectorSampleConfig
extends ConnectorConfig
implements CVComponentConfigSamplify {
    @IsMandatory(true)
    public void setPrimaryServer(final WebServer primaryServer) {
        this.primaryServer = primaryServer;
    }
    public WebServer getPrimaryServer() {
        return this.primaryServer;
    }
    @IsMandatory(false)
    public void setSecondaryServers(final WebServer[] secondaryServers) {
        this.secondaryServers = secondaryServers;
    }
    public WebServer[] getSecondaryServers() {
        return this.secondaryServers;
    }
    @Override
    public void samplify() {
      // samplify is called after the config instantiation.
      this.primaryServer.setServer("mywebserver.mydomain");
      this.primaryServer.setPort(80);
```

```
      // Thus there will be a default value for the primary server name
      // but it won't appear when adding a new secondary server
    }
      // a webserver is made of a server name and a network port
      public static class WebServer {
        @IsMandatory(true)
        public void setServer(final String server) {
            this.server = server;
        }
        public String getServer() {
            return this.server;
        }
        @IsMandatory(true)
        public void setPort(final int port) {
            if (port < 0 || port > 65535) {
                final ConfigurationException e =
                        new ConfigurationException("Invalid network port: " + port);
                e.setConfigKey("Port");
                throw e;
            }
            this.port = port;
        }
        public int getPort() {
            return this.port;
        }
        public static String[] getMethods() {
            return new String[] {
                "Server",
                "Port"
            };
        }
        private String server;
        private int port;
    }
    private WebServer primaryServer = new WebServer();
    private WebServer[] secondaryServers;
    public static String[] getMethods() {
        return new String[] {
            "PrimaryServer",
            "SecondaryServers"
        };
    }
}
```

The UI should be similar to the following screenshot:



Setter/Getter methods

This section describes the setter and getter methods when packaging your CVPlugin.

About setters

Setters are:

- public methods that do not return any value (`void` return type),
- whose names are prefixed by `set` (the property names following the `set` prefix keep the same letter case),
- and take exactly one argument (the value to be set).

Setters may bear annotations, see Setter function annotations. Setters annotated with the `IsHidden (com.exalead.config.bean.IsHidden)` class are ignored.

These are the setters for the `Proxy` and `ProxyPort` properties.

```
@IsMandatory(false)
@PropertyLabel("Proxy to use")
public void setProxy(final String proxy) {
    this.proxy = proxy;
}
@IsMandatory(false)
@PropertyLabel("Proxy port to use")
public void setProxyPort(final int port) {
    this.port = port;
}
```

Note: The first upper case is kept for the property name.

Each setter method should be associated with a getter method.

About getters

Getters are:

- public methods returning a value,
- the returned value should be equivalent to the value taken as argument in the setter,
- and take no argument.

Getters do not bear any annotation.

These are the getters for the `Proxy` and `ProxyPort` properties.

```
public String getProxy()
{
    return proxy;
}
public int getProxyPort()
{
    return proxy;
}
```

Note: The first upper case is kept for the property name.

The available types, recognized by the framework for setters and getters are the following ones:

- The String class.
- Base boolean types: boolean, Boolean.
- Base numerical types: byte, char, short, int, long, float, double, Boolean, Byte, Character, Short, Integer, Long, Float, Double.
- Any class supporting the public static `valueOf` method taking exactly one String argument, and returning an object of its own class (in this case, the `toString` method from the Object base class will also be used); such as `enums` classes.
- Any class taking exactly one String argument (in this case, the `toString` method from the Object base class will also be used); such as the `Date` class.
- Generic Interface class (to accept outer components in configurations; used internally only).
- An array of an allowed type.

Example of array parameters:

```
public class NotesServerConfig implements CVComponentConfigSamplify {
     public static class NotesDBRule
```

```
    {
        public NotesDBRule(final String rule)
        {
            this.rule = rule;
        }
        public NotesDBRule()
        {
        }
        protected String rule;
        public String getRule()
        {
            return rule;
        }
        @Override public String toString() {
            return getRule();
        }
    }
    private NotesDBRule[] includes = new NotesDBRule[] {};
    @IsMandatory(false)
    @PropertyLabel("Include rules (regular expression)")
    public void setIncludes(final NotesDBRule[] includes) {
        this.includes = includes;
    }
    public NotesDBRule[] getIncludes() {
        return includes;
    }
}
```

Setter function annotations

The following annotation classes may be used in setters to provide additional information on property types and settings.

Annotation	Description
`@BeeKeyValueType (com.exalead.config.bean.BeeKeyValu`	Possible values include: • `string` for generic string, • `numeric` for a signed long integer value, • `enum:value1,value2...` for a selection restricted to certain values. • `encrypted` for values that shall be encrypted, such as passwords.

Annotation	Description
	• `hidden` for values that must not be displayed/ edited at all. Consider using `@IsHiddenUI` instead.
`@Connector (com.exalead.config.bean.Connector)`	Displays a combo box to list configured connectors. The list of connectors can be restricted with the following flags: • `allowSelf`: if the component describes a connector, it indicates whether this connector is included in the combobox (default is false). • `allowDeployed`: indicates whether deployed connectors are listed (default is true). • `allowUndeployed`: indicates whether undeployed connectors are listed (default is true) • `allowUnmanaged`: indicates whether unmanaged connectors are listed (default is true) • `allowManaged`: indicates whether managed connectors are listed (default is true) • `allowedClassId`: a regular expression to filter by connector classid. If the regular expression is an empty string, then all connectors are listed. • `forbiddenClassId`: a regular expression to exclude connectors by classid. • `allowEmpty`: adds an extra empty option in the combo box. Property value will be "" when this option is selected.
`@DataModelClass (com.exalead.config.bean.DataModelC`	Displays a combo box to list available classes in all configured data models. The data models to search classes in can be restricted with the following options (both options are mutually exclusive): • `dataModel`: only list classes of the given data model. • `buildGroup`: only list classes of the data model referenced by the given build group.

Annotation	Description

`@Date (com.exalead.config.bean.Date)`	Displays a calendar to configure the property field. The property field will be stored as a string formatted with the date format specified in the annotation.
	Default date format is: `@Date(format="yyyy-MM-dd")`
	Example: If you add a `startDate` field with
	`@Date(format="yyyy-MM-dd")`
	`public void setStartDate(String startDate) {`
	`this.startDate = startDate;`
	`}`
	The UI will display: **Start date**: [calendar widget]
	And when dates are saved, they will be stored with the `yyyy-MM-dd` date format.

`@DateTime (com.exalead.config.bean.DateTime)`	Same as `@Date` with time information. Default datetime format is: `@DateTime(format="yyyy-MM-dd HH:mm:ss")`

`@EnumFieldType (com.exalead.config.bean.EnumFieldT`	The given setter takes only a subset of String representations (enum) as value.
	Example:
	`@EnumFieldType(possibleValue = { @PossibleValueType("red"),`
	`@PossibleValueType("green"), @PossibleValueType("blue") })`
	Note: in `@PossibleValueType` you can associate a label to each value.

`@IsHidden (com.exalead.config.bean.IsHidden)`	The given setter is to be ignored. For example, you can use it if the function is not part of the bean setter subset. Example: `@IsHidden()`

Annotation	Description
`@IsHiddenUI (com.exalead.config.bean.IsHiddenUI`	The given setter property should not be editable or displayed, but must still be processed as a regular property. Use it to hide internal properties. Example: `@IsHiddenUI()`
`@IsMandatory (com.exalead.config.bean.IsMandator`	If the associated boolean is `true`, then the given setter property must be defined (non-empty unserialized value) so that the configuration can be considered as valid. Otherwise, the property will be considered as optional. Note: Without this annotation, the default is mandatory. Example: `@IsMandatory(false)`
`@MultiLineString (com.exalead.config.bean.MultiLineS`	Displays a multi-line text input control.
`@Path (com.exalead.config.bean.Path)`	Displays a file chooser widget, that browses Exalead CloudView host filesystem.
`@PropertyDescription (com.exalead.config.bean.PropertyDe`	The property description, as a string. Typically displayed for comment or tooltip. It supports text only, no HTML code. Example: `@PropertyDescription("Define the hostname to be used for the proxy")`
`@PropertyLabel (com.exalead.config.bean.PropertyLa`	The property label, as a string. Typically displayed for short name. Example: `@PropertyLabel("Proxy hostname")`
`@SecuritySource (com.exalead.config.bean.SecuritySc`	Displays a combo box to list configured security sources. The list of security sources can be restricted with the following flags:

Annotation	Description
	• `allowSelf`: if the component describes a security source, indicates whether this security source is included in the combobox (default is false) • `allowDeployed`: indicates whether deployed security sources are listed (default is true) • `allowUndeployed`: indicates whether undeployed security sources are listed (default is true) • `allowedClassId`: a regular expression to filter by security source classid. If the regular expression is an empty string, then all security sources are listed. • `forbiddenClassId`: a regular expression to exclude security sources by classid. • `allowEmpty`: adds an extra empty option in the combo box. Property value will be "" when this option is selected.

`@WithSuggest (com.exalead.config.bean.WithSugges`	Displays a text input that performs remote queries to suggest entries. In order to remote suggestions to work, the CVComponent must support `SuggestOption` queries. Here is a full example:
	Component config class:
	`MyComponentConfig extends ConnectorConfig {`
	` ...`
	
	` private String folder;`
	
	` @WithSuggest`
	` public void setFolder(String folder) {`
	`  this.folder = folder;`
	` }`
	`}`
	Component class:
	`@IntrospectableComponent(introspectorClass=MyCo`
	`supportedQueries = {@SupportedQuery(queryClass`

Annotation	Description

	`class MyComponent extends Connector { ... }`
	Introspector class:
	`class MyComponentIntrospector implements CVComp public Object execute(CVComponentConfig compon if (query instanceof SuggestOption) { SuggestOption suggestQuery = (SuggestOption) SuggestOption.SuggestResult result = new Sugg // disambiguate by field name if you have s if ("Folder".equals(suggestQuery.getConfigKey String currentValue = suggestQuery.getCurren MyComponentConfig myConfig = (MyComponentCon String[] suggestedValues = ...; result.setSuggestedValues(suggestedValues); } return result; } throw new IllegalStateException("Unknown int } }`

Available classes for setters and getters

The following classes can be used in setters and getters.

Class	Description
`BooleanChecked (com.exalead.config.bean.BooleanChe`	Ensures that the value entered for an option is either `true` or `false`. Use it with boolean options to enforce the option values to `true` or `false`.
`BytesValue (com.exalead.config.bean.BytesValue`	A value representing a number of Bytes, in any SI Unit. For example, you can use "10KB" or "10MB" as value, and still handle an amount of bytes in your code.
`MillisecondsValue (com.exalead.config.bean.Millisecor`	A value representing a number of milliseconds, in any SI Unit.

Class	Description
	For example, you can use "2min" or "5s" as value, and still handle an amount of milliseconds in your code.
`OptionalBoolean (com.exalead.config.bean.OptionalBc`	A value representing a tri-state Boolean value, with checked value (`true`, `false` or `optional`)
`KeyValue (com.exalead.config.bean.KeyValue)`	A value representing a pair of strings (key and a value).

Possible Setter exceptions

A setter may raise exceptions, such as `NumberFormatException` or `IllegalArgumentException`, but we strongly recommend using the following built-in exceptions.

Recommended exception	Description
`ConfigurationException (com.exalead.config.bean.Configurat`	Generic exception thrown when a configuration exception occurs.
`IllegalValueException (com.exalead.config.bean.IllegalVal`	Exception thrown when a setter encounters an illegal value.

Implementing Format Plugins

The `Text Extractor (all mime types)` component is used in the analysis to extract text and metadata from various file types (such as Office files, PDF, etc.). A similar component is also used to produce HTML previews from the same set of file types, and generate thumbnails when displaying results.

This system can be extended using the plugins mechanism, to support more file types for extraction at indexing time, but also for HTML preview and thumbnails calculation.

Technical Overview

A *format plugin* is a regular plugin and its main component can have an associated configuration object. For more information, see Packaging the connector as a plugin.

The format component must implement:

- `DocumentPartTransformer (com.exalead.pdoc.plugins.DocumentPartTransformer)`

- and, as usual, `CVComponent` (`com.exalead.mercury.component.CVComponent`).

It must provide one of the following constructors:

- Either a default constructor (taking no argument), if the component has no associated configuration.
- Or, as usual, a constructor taking an object implementing `CVComponentConfig`. In that case, a `CVComponentConfigClass` annotation must be present. See Top level component class(es).

  Note: You may want to extend the `DocumentPartTransformer.DefaultImpl` class `(com.exalead.pdoc.plugins.DocumentPartTransformer.DefaultImpl)` which already provides default methods.

Two methods must be implemented as described in the following sections.

First method

The first method allows you to advertise the supported output MIME types, that is to say, the MIME type of data produced after transformation, for the two kinds of transformation (extraction of data, or display).

```
/**
  * Get the list of supported MIME output formats per transformation kind.
  * For TransformKind.display, the types advertise the first produced part
  * MIME type. (an hypertext text/html document may have additional parts
  * with different formats)
  *
  * @returns an empty array if the transformation is not supported
  * @param kind
  * the kind of transformation: Note: the returned array may
  * include Format.MIME_GENERIC to advertise a filter producing
  * any type of document or an unknown subset of document types.
**/
     public List<String> getSupporterOutputMime(TransformKind kind);
```

Example

If the component is able to produce plain text when extracting, and HTML for display, the code can be:

```
public List<String> getSupporterOutputMime(TransformKind kind)
     {
         final List<String> list = new ArrayList<String>();
         if (kind == TransformKind.extract) {
             list.add("text/plain");
         } else if (kind == TransformKind.display) {
             list.add("text/html");
```

```
        }
        return list;
    }
```

Second method

The second method will process the input document, and produce extracted data or previews.

```
/**
  * Transform a part 'part' using the format 'format' into a destination
  * document. Use getSupporterOutputMime() to get the list of supported
  * output MIME types.
  *
  * @param part
  *        the input part.
  * @param format
  *        the transformation format
  * @param input
  *        the source document (part is probably in this document)
  * @param output
  *        the target document (parts are stored inside this document)
  * @throws UnsupportedInputFormatException
  *        if the input format is not supported by the filter
            (in this case, the upstream client will give up with the current filter)
  * @throws UnsupportedOutputFormatException
  *        if the output format is not supported by the filter
            (in this case, the upstream client may retry with a different format, usin
  * @throws NoSuchMethodException
  *        if the method is not supported
  * @throws TransformationException
  *        upon error (in this case, the upstream client may choose to
  *        give up on the input, or select another filter)
  *        Note: input and output may be the same objects
**/
  public void transform(DocumentPart part, ProcessableDocument input,
          ProcessableDocument output, Format format)
          throws TransformationException, UnsupportedInputFormatException,
          UnsupportedOutputFormatException, NoSuchMethodException;
```

This function takes:

- a part as input (the part contains data, and associated metadata),
- the related input document, which is generally unused,
- the output document where multiple parts might be added,
- and the requested format (whether information extraction is requested for the display processing and the output MIME type).

When producing content:

- Each produced file must be embedded in a Part document, created through the output object `addPart` method.
- For referenced parts, parts must have proper MIME type advertised, and proper filenames. If the first HTML part embeds relative links to resources, the given resources must be properly named, using the same relative filenames.

  Note: The part name is usually `preview` for a preview, and `document` for extracted metadata, but the naming is free. The first part must be the leading part. However, if the produced content is an HTML preview, the first part must be the master document.

The following example shows the skeleton of a `transform()` method:

```
@Override
public void transform(DocumentPart part, ProcessableDocument input,
         ProcessableDocument output, Format format)
         throws TransformationException, UnsupportedInputFormatException,
         UnsupportedOutputFormatException
{
     // Validate requested output format
     final boolean isText = format.getMime().equalsIgnoreCase(
             Format.MIME_TEXT);
     final boolean isHtml = format.getMime().equalsIgnoreCase(
             Format.MIME_HTML);
     final String outMime = format.getMime();
     if (!isText && !isHtml) {
         throw new UnsupportedOutputFormatException("unsupported format");
     }

    // Validate input format
     String mime = part.getComputedMime();
     if (!isNotMyFormat(part.getFilename(), part.getForcedMime())) {
         throw new UnsupportedInputFormatException("unsupported MIME type");
     }
     // Transform
     try {
         byte[] data = part.getContentAsBytes();
...
         if (isHtml) {
             // Prepare final part
             final DocumentPart dp = output.addPart("preview");
             dp.setEncoding("utf-8");
             dp.setForcedMime(format.getMime());
...
             dp.setContent(xml.toString().getBytes("UTF-8"));
         }
```

```
    } catch (IOException io) {
        throw new TransformationException(io);
    }
}
```

Extending the Files Connector through Plugins

The filesystem connector embeds natively a number of schemes and protocols: native filesystem, Windows share filesystem (\\path or smb:// URLs), basic ftp support (ftp:// URLs), basic http (http://), etc.

It is possible to extend the features of the filesystem connector and use all the embedded features of the connector (multithreaded scan, containers handling, etc.) without having to create a new connector, by implementing additional protocol schemes through plugins.

Note: For a description of the Files Connector features, see "Files Connector" in the Exalead CloudView Connectors Guide.

Technical Overview

A filesystem connector interface plugin is a regular plugin, providing a factory component without any associated configuration.

The main filesystem connector interface component must implement `FileInterfaceFactory (com.exalead.papi.connectors.filesystem.FileInterfaceFactory)` and, as usual, `CVComponent (com.exalead.mercury.component.CVComponent)`. It must provide a default constructor (no arguments).

Two methods must be implemented.

First Method

The first method allows you to define supported root path schemes, that is to say, whether the root path is recognized by this plugin.

```
/**
  * Test whether a root path is handled by this factory;
  * i.e. if build() may be called upon this path.
  * The factory needs to ensure the namespace used will not conflict with
  * any native namespace, of with previous plugin.
  *
  * @param rootConf
  *            The root path.
  * @return true if the root path is handled by this factory.
*/
```

```
  public boolean canHandle(final FilesystemRootPathConfig rootConf);
```

This method must return `true` if the root path passed is recognized by the plugin. It will typically check the syntax of `rootConf.getRootKey()` against a known specific URL scheme prefix.

Important: Make sure that no other plugin is using this prefix, or the filesystem connector will raise an error due to the namespace conflict.

For example, when using "`myfile://`" as prefix for root paths, you may use:

```
@Override
public boolean canHandle(FilesystemRootPathConfig rootConf) {
    final boolean handle = rootConf.getRootKey().startsWith("myfile://");
         return handle;
}
```

Second Method

The second method will provide an instance of `FileInterface (com.exalead.papi.connectors.filesystem.FileInterface)` to handle the virtual underlying filesystem. This method will only be called by the framework if `canHandle()` returned `true` upon the same configuration object.

```
/**
  * Build a new FileInterface
  *
  * @param rootConf
  *            the root path
  * @return The FileInterface
  * @throws IOException
  *             Upon I/O error during object creation
  * @throws IllegalArgumentException
  *             If the root path is unsupported (ie. canHandle() would have
  *             returned false)
  */     public FileInterface build(final FilesystemRootPathConfig rootConf) throws I
IllegalArgumentException;
```

The object passed provides the root key `(getRootKey())` and authentication details if needed.

For example, when using "`myfile://`" as prefix for root paths, you may use:

```
@Override
public FileInterface build(FilesystemRootPathConfig rootConf) throws IOException, Ill
         if (!canHandle(rootConf)) { // unexpected
             throw new IllegalArgumentException("unsupported scheme");
         }
         final File root = new
File(rootConf.getRootKey().replace("myfile://", ""));
         return new MyFileInterface(root);
}
```

The `FileInterface (com.exalead.papi.connectors.filesystem.FileInterface)` interface provides the necessary functions to handle a virtual filesystem (listing the directory, opening a file, fetching attributes, etc.):

```
package com.exalead.papi.connectors.filesystem;
import java.io.IOException;
import java.util.Iterator;
import com.exalead.papi.helper.Meta;
import com.exalead.papi.helper.stream.ContentStreamSafe;
/**  * Abstract file interface.  */
public interface FileInterface {
     /**
      * Get the absolute path.
      *
      * @return The absolute path.
      */
     public String getAbsolutePath();
     /**
      * Is the file a file ?      *
      *      @return true if this is a file
      */
     public boolean isFile();
     /**
      * Is the file a directory ?
      *
      * @return true if this is a directory
      */
     public boolean isDirectory();
     /**
      * Is the file a link ?
      *
      * @return true if this is a link
      */
     public boolean isLink();
     /**
      * Get the path leaf name.
      *
      * @return the path leaf name
      */
     public String getName();
     /**
      * Last-modified date.
      *
      * @return Last-modified date, or 0 if not supported.
      */
     public long lastModified();
     /**
      * Return the time when the file was last accessed (in milliseconds since
```

```
             * Epoch)
             *
     * @return Last-access date, or 0 if not supported.
     **/
    public long lastAccess();
    /**
     * Return the time when the file was created (in milliseconds since Epoch)
     * Return 0 if this attribute if unsupported by the filesystem.
     *
     * @return Creation date, or 0 if not supported.
     **/
    public long creation();
    /**
     * The file length.
     *
     * @return file length
     */
    public long length();
    /**
     * Does the file exist?
     *
     * @return true if the file exists
     */
    public boolean exists();
     /**
     * Is the file readable?
     *
     * @return true if the file is readable
     */
    public boolean canRead();
    /**
     * Get security meta-data.
     *
     * @return security meta-data
     */
    public Meta[] getSecurityMetas() throws IOException;
    /**
     * Get additional meta-data.
     *
     * @return additional meta-data, or @c null if no additional meta-data are
     *      present.
     */
    public Meta[] getAdditionalMetas() throws IOException;
     /**
     * Get contents.
     *
     * @return The stream contents.
     * @throws Exception
```

```
     *      Upon error.
     */
    public ContentStreamSafe getContents() throws Exception;
     /**
     * Enumerate files. Only available for directories.
     *
     * @return the iterator
     */
    public Iterator<FileInterface> enumerateFiles(FileInterfaceFilterfilter) throws
    /**
     * Enumerate files. Only available for directories.
     *
     * @return the iterator, or @c null upon error
     */
    public Iterator<FileInterface> enumerateFiles() throws IOException;
     /**
     * Get a child.
     *
     * @param name
     *      The child name.
     * @return the child.
     */
    public FileInterface getChild(String name); }
```

Example of a very basic implementation of a filesystem scheme (this sample is available in the sample list):

```
public class MyFileInterface implements FileInterface {
    protected final File file;
    public MyFileInterface(File file) {
        this.file = file;
    }
    @Override
    public boolean canRead() {
        return file.canRead();
    }
    @Override
    public long creation() {
        return -1; // unsuppoorted
    }
    @Override
    public Iterator<FileInterface> enumerateFiles(FileInterfaceFilterfilter) throws
        final List<FileInterface> list = new ArrayList<FileInterface>();
        for (final File f : file.listFiles()) {
            final MyFileInterface child = new MyFileInterface(f);
            if (filter == null || filter.accept(child)) {
                list.add(child);
            }
        }
```

```
        return list.iterator();
    }
    @Override
    public Iterator<FileInterface> enumerateFiles() throws IOException {
        return enumerateFiles(null);
    }
    @Override
    public boolean exists() {
        return file.exists();
    }
    @Override
    public String getAbsolutePath() {
        return file.getAbsolutePath();
    }
    @Override
    public Meta[] getAdditionalMetas() throws IOException {
        return new Meta[] { new Meta("canonical_path",file.getCanonicalPath()) };
    }
    @Override
    public FileInterface getChild(String name) {
        return new MyFileInterface(new File(file, name));
    }
    @Override
    public ContentStreamSafe getContents() throws Exception {
        return new MyContentStreamSafe(file);
    }
    @Override
    public String getName() {
        return file.getName();
    }
    @Override
    public Meta[] getSecurityMetas() throws IOException {
        return new Meta[] { SecurityMeta.getPublicSecurityMeta() };
    }
    @Override
    public boolean isDirectory() {
        return file.isDirectory();
    }
    @Override
    public boolean isFile() {
        return file.isFile();
    }
    @Override
    public boolean isLink() {
        return false;
    }
    @Override
    public long lastAccess() {
```

```
            return -1; // unsuppoorted
        }
        @Override
        public long lastModified() {
            return file.lastModified();
        }
        @Override
        public long length() {
            return file.length();
        }
}
```

Developing a Security Source

Describes how to develop a security source for your custom managed connector

About Security Source Development

Security sources are used to manage security information relative to users, or group of users.

The main goal of security sources is to:

- authenticate a user (using its password) and return its security identifiers, called tokens,
- list security tokens associated with a given user or group.

When a document is produced by a connector, the `security` metadata pushes the list of tokens which give the required access credentials to the indexed document.

Negative tokens can also be used to refuse credentials. In such case, negative rules are always prioritary, that is to say that if a positive token gives access to a document, and a negative one denies it, the access will be denied.

By default, all security tokens are indexed in the product, to enable security features per document.

Users also have a set of similar tokens associated with their authenticated accounts. These tokens are usually based on their access rights or group ownership.

An authenticated user will only be able to find a document, if his set of security tokens contains at least an allowed token, and no negative token.

Connectors and security sources work together, the tokens produced by the former are compared to the later to reduce the search results scope.

For example:

A filesystem source connector produces the following tokens (the `security` meta-data will contain these values):

- `unix:user:10028`
- `unix:group:100`

Any authenticated user whose token contains either `unix:user:10028` or `unix:group:100` will therefore have access to the document.

Implementing a Security Source Plugin

A security source is a regular plugin with an associated configuration object.

Its design is quite similar to the Connector one. The security source class must implement the `SecuritySource (com.exalead.security.sources.common.SecuritySource)` interface, and must define a constructor taking a configuration class.

Implement the Security source part

```
@CVComponentConfigClass(configCheckClass = CVComponentConfigCheckNone.class,
configClass =LocalSecuritySourceConfig.class)
@CVComponentDescription("Local Security (generic)")
 public class LocalSecuritySource extends SecuritySource implements CVComponent {
     public LocalSecuritySource(LocalSecuritySourceConfig config) {
     ...
     }
     ...
}
```

Implement the Associated config part

```
@CVComponentDescription("Local Security (generic)")
@IsEmptyConfig(true)
public class LocalSecuritySourceConfig implements CVComponentConfig {
     ...
}
```

Implement the security source methods

The following methods must be implemented within the security source.

Method	Description
`public abstract AuthenticationResult authenticate(String login, String password, boolean needPassword) throws SecurityException;`	This method authenticates a user and returns authorizations, such as success status, security tokens and associated information, with: • the `login` login name, • and an optional credential `password` to check if `needPassword` is set to `true` Otherwise, the function always returns a valid object which can be used to list the user security tokens.

Method	Description
`public List<String> getUsers() throws Exception;`	Lists all users contained in the security source. It may return an empty list if such information is not available.
`public List<String> getGroups() throws Exception;`	Lists all groups contained in the security source. It may return an empty list if such information is not available.
`public SecurityToken getUserToken(String user);`	Gets the security token list of a user.
`public SecurityToken getGroupToken(String group);`	Gets the security token list of a group.

Implement the AuthenticationResult class

The returned `AuthenticationResult (com.exalead.security.sources.common.AuthenticationResult)` object should be filled using the following methods.

Method	Description
`public void setSuccess(Boolean value);`	If authentication was requested, it sets the success result.
`public void setCause(String value);`	If authentication was requested and failed, it provides the error description.
`public void setUserId(String value);`	Sets the user identifier.
`public void setUserDisplayName(String value);`	Sets the user display name, usually its first and last names.
`public void setSecurityTokens(List<SecurityToke tokens);`	If no authentication was requested, or if the authentication was successful, it provides the list of security tokens owned by the user.

Example:

```
List<SecurityToken> tokens = new ArrayList<SecurityToken>();
 tokens.add(new SecurityToken("unix:user:10028"));
 tokens.add(new SecurityToken("unix:group:100"));
AuthenticationResult results = new AuthenticationResult();
 results.setSuccess(true); results.setUserId("10028");
```

```
results.setUserDisplayName("John Doe");
results.setSecurityTokens(tokens);
```

Implement the SecurityToken class

Each security token is returned inside a `SecurityToken (com.exalead.security.sources.common.SecurityToken)` object. Its constructor takes the security token string as sole argument.

Example:

```
SecurityToken st = new SecurityToken("unix:user:10028");
```

Deploying the Connector

Describes how to deploy the connector plugin in Exalead CloudView and how to configure it in the Administration Console

Deploying the Connector Plugin

Your custom connector must be packaged as a plugin to be deployed in Exalead CloudView.
You can deploy your plugins using either the Administration Console or the `cvadmin` tool (on the command line).

Install a plugin in the Administration Console

1. Go to **Deployment > Plugins**
2. Click **Upload plugin** and browse for your file.

Install a plugin on the command line

You can install plugins regardless of whether the Exalead CloudView product is running or stopped.

1. Go to `<DATADIR>/bin` and run: `./cvadmin plugins`
2. Enter: `install file=myplugin.zip`

   Note:

   For multi-host installs, you must run this command on the master host. The plugin will automatically be distributed to all hosts.
3. Restart the processes for which you are going to use its components, typically the search-server or analyzer.

You can then use the plugin.

List installed plugins

1. From the `<DATADIR>/bin`, get the list of the installed Exalead CloudView plugins: `./cvadmin plugins list`

Uninstall a plugin

1. From the `<DATADIR>/bin` get the list of the installed Exalead CloudView plugins: `./cvadmin plugins list`
2. Remove the plugin: `./cvadmin plugins remove name=myplugin`

Maintaining a Connector Configuration across Versions

The following code sample shows how to implement the `upgrade-config` capability in your connector.

For more information about this command, see "Upgrade a connector" in the Exalead CloudView Administration Guide.

```
package com.exalead;
import com.exalead.mercury.component.*;
import com.exalead.mercury.component.config.CVComponentConfig;
import com.exalead.papi.framework.connectors.Connector;
import com.exalead.papi.framework.connectors.ConnectorConfig;
import com.exalead.papi.framework.connectors.introspection.UpgradeConfig;
import exa.bee.KeyValue;
@IntrospectableComponent(
    // register the {@link MyUpgradableConnector.Introspector} class to handle intros
    introspectorClass=MyUpgradableConnector.Introspector.class,
    // tells CloudView this connector supports the config upgrade capability
    supportedQueries={ @SupportedQuery(queryClass=UpgradeConfig.class)})
public class MyUpgradableConnector extends Connector {
    public MyUpgradableConnector(final ConnectorConfig config) throws Exception {
        super(config);
    }
    public static class Introspector implements CVComponentIntrospector {
        // Member method called to process introspection queries.
        @Override
        public Object execute(
                final CVComponentConfig componentConfig,
                final IntrospectionQuery query) throws Exception {
            if (query instanceof UpgradeConfig) {
                final UpgradeConfig up = (UpgradeConfig) query;
                System.out.println("Updating configuration of connector " + up.getCon
        //Specify the component versions explicitly and the dependencies to perform t
        //The following example shows how to upgrade from version 1.0 to 2.0 if you n
        // an intermediary upgrade to version 1.1
        //You can either upgrade from 1.0 to 2.0 OR from 1.1 to 2.0
        //CAUTION: As by default CloudView is not able to provide the previous connec
        //to this method, if you choose to migrate from 1.1 to 2.0, the same code wil
```

```
        //In our example, it will execute the upgrade operation starting from 1.0.
               applyChangesFrom_1_0To1_1(up.getCurrentConfig());
                applyChangesFrom_1_1To2_0(up.getCurrentConfig());
                return up.getCurrentConfig();
            }
            return null;
        }
        void applyChangesFrom_1_0To1_1(final KeyValue config) {
            renameKey(config, "Foo", "Bar");
        }
        void applyChangesFrom_1_1To2_0(final KeyValue config) {
        // [...] apply required changes. For example, if version 2.0 is multithreaded
        // you need to set up the threadPoolSize property...
        }
        /**
          * Recursively look for a key in the connector configuration, and rename it
          * @param config The configuration to upgrade
          * @param prevKey The configuration key name to replace
          * @param newKey The new key value
          * @return The updated configuration
          */
        public KeyValue renameKey(final KeyValue config, final String prevKey, String
            if (config.getKey() != null) {
                if (config.getKey().equals(prevKey)) {
                    config.setKey(newKey);
                }
            }
            for (int i = 0; i < config.getKeyValue().size(); ++i) {
                config.getKeyValue().set(i, renameKey(config.getKeyValue().get(i), pr
            }
            return config;
        }
    }
}
```

Creating and Configuring the Connector

Once you have implemented and installed your connector, it should be available in the list of connectors displayed in the **Type** property of the **Add Connector** dialog box.

1. Go to the Administration Console.
2. In **Index > Connectors**, click **Add connector** and enter a name for your custom connector. For example, `Basic filesystems`
3. Select your connector type from the drop-down list and click **Accept**.

4. In your connector's **Configuration** tab, click **Add Entry** to add the additional properties to the **Global Config**.
5. Click **Apply** to apply the configuration.

Advanced Operations and Best Practices

This chapter describes several considerations that should be taken into account when developing your own connector.

What to map from the Data Source?

In a way, indexing can be seen as creating a mapping function between objects from the data source, to documents in the index. While this mapping may seem obvious at first, the question shouldn't be overlooked, as it structures the behavior of the search engine.

There is not always a 1 to 1 mapping between unit objects in the data source, and documents in the index.

For example, suppose you are writing a connector for a data source dealing with emails. Should it be possible for a user to find emails based on the content of their attachments? Most probably yes, therefore this connector is probably going to map an email and all its attachments with a single document.

Should it also be possible to find a whole thread of discussion, query with quotes from an email? If so, then the connector will probably push along with the previous documents, 1 document per thread, in which the content of all emails will have been mapped.

For example:

- For emails / forums: To find a thread, you could have:
  - 1 Email = 1 Document
  - All emails belonging to the same thread = 1 Document
- Enovia

- 1 object made of several parts = 1 document

- Database
  - Star or snowflake schema join = 1 document

    Note: To aggregate data this way, you can use the Consolidation Server.

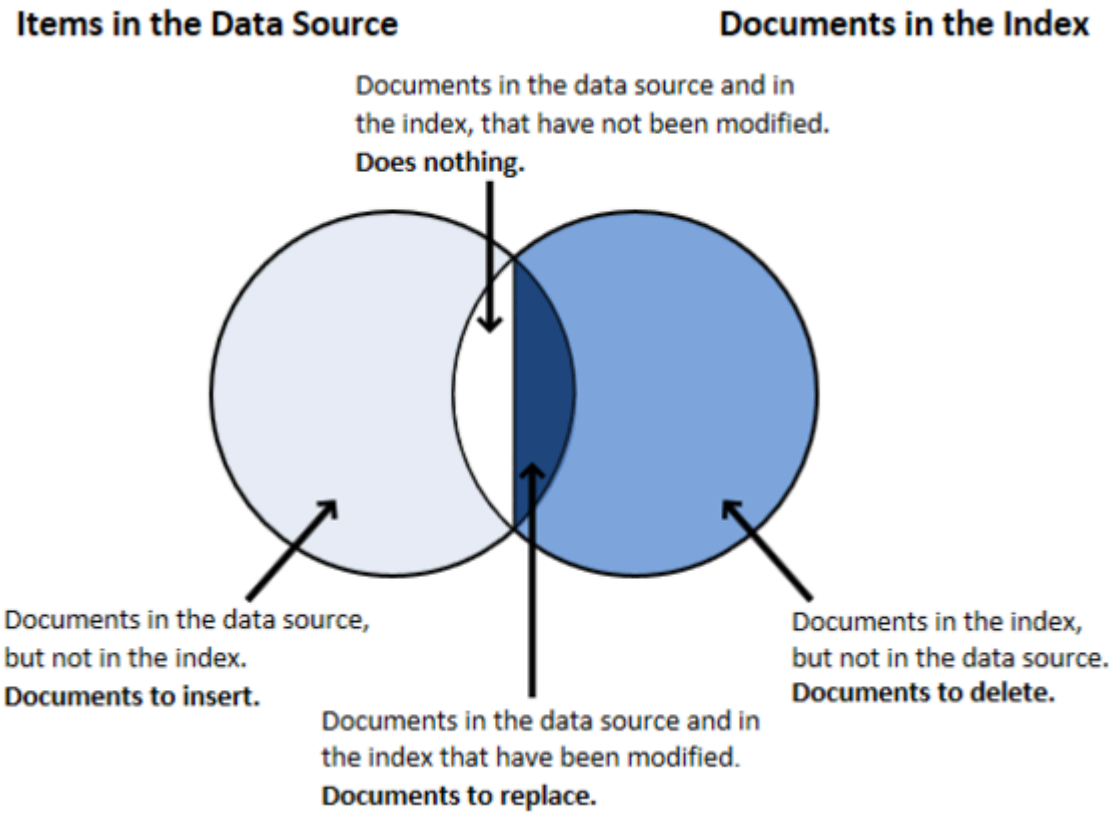
How to Keep the Index Synchronized with the Datasource

Strategy 1: The full scan approach

The easiest strategy is to define a single function, which every time it is called, triggers a full scan on the data source, and pushes all documents found in the data source to the index.

Strategy 2: The differential approach

An improvement to the first strategy is to only push the differences between the contents of the data source and the index. These differences can be described using four categories, as shown in the schema below.



Implementing Synchronization

When indexing a document collection that is evolving, the task of your connector is to ensure that the state of the index always follows the state of the source. This includes detecting new documents, modified documents and deleted documents.

Exalead CloudView provides two mechanisms to help implementing synchronization.

Stamp-based synchronization

In Exalead CloudView, each document has a stamp, which is an opaque String. When you push a document with a stamp, the stamp is stored and can be retrieved. This can be used to detect whether a document has been modified since its last push. For example, you could set the "last modification timestamp" of the document as the stamp, or its MD5 hash.

A basic example would look like:

```
for (Document document : listDocumentsInDataSource()) {
  String currentStamp = computeStamp(document);
  DocumentStatus statusInCloudView = papi.getDocumentStatus(document.getURI());
    if (statusInCloudView == NOT_PRESENT) {
    /* This document is not in CloudView, so it's new in the data source -> push it *
    papi.addDocument(document);
  } else {
    if (!statusInCloudView.stamp.equals(currentStamp)) {
      /* Stamp has changed: the document was modified -> push it */
      papi.addDocument(document);
    }
  }
}
```

However, this method has two drawbacks:

- Calling the `getDocumentStatus()` method for each document is slow, as it involves one synchronous PAPI call for each document.
- It does not handle documents to be deleted in Exalead CloudView.

To fix this, you can list all documents in Exalead CloudView, and compute the differences. For example:

```
Map<String, String> stampsOfDocumentsInCloudView;
for (SyncedEntry se : papi.enumerateSyncedEntries()) {
  stampsOfDocumentsInCloudView.put(se.getURI(), se.getStamp());
}
Set<String> documentsInDataSource;
for (Document document : listDocumentsInDataSource()) {
```

```
  documentsInDataSource.add(document.getURI());
  String currentStamp = computeStamp(document);
  String stampInCloudView = stampsOfDocumentsInCloudView.get(document.getURI());
    if (stampInCloudView == null) {
    /* This document is not in CloudView, so it's new in the data source -> push it *
    papi.addDocument(document);
  } else {
    if (!stampInCloudView.equals(currentStamp)) {
      /* Stamp has changed: the document was modified -> push it */
      papi.addDocument(document);
    }
  }
}
/* Now, compute the list of deleted documents: documents that are in CloudView but nc
for (String docInCloudView : stampsOfDocumentsInCloudView.keySet()) {
  if (!documentsInDataSource.contains(docInCloudView)) {
     /* Doc is not in data source anymore -> Delete it from !CloudView */
     papi.deleteDocument(docInCloudView);
  }
}
```

This method might not be convenient if you have huge amounts of documents in Exalead CloudView, due to the large memory requirements to store the list.

Other possible method enhancements are:

- Batch enumeration over known data subsets. For example, folders in a filesystem connector.
- Parallel enumeration – if suitable, this enumerates both the data source and Exalead CloudView in parallel. The Exalead CloudView enumeration is guaranteed to be in lexicographical order based on the URIs. As the Exalead CloudView enumeration is streamed, you can perform a merge between the lists and compute the differences on the fly.

Checkpoint-based synchronization

Stamp-based synchronization is generally quite costly due to the memory requirements and should only be used when there is no notion of "event log" in the source. Many data sources have logs or mechanisms to determine what has changed between two events. In this case, you should use checkpoint-based synchronization.

A checkpoint is an opaque String, not associated with a document, that is stored persistently by Exalead CloudView, and can be retrieved.

If a checkpoint can be retrieved, then all operations that were sent to the PAPI before the checkpoint are guaranteed to be safely stored to disk, and will never be lost, even if they are not yet searchable.

The following sample shows the workflow of a checkpoint-based synchronization:

```
final static String CHECKPOINT_NAME = "my_checkpoint";
public void syncSource() {
  String currentCheckpointValue = papi.getCheckpoint(CHECKPOINT_NAME);
  String currentLast = dataSource.getCurrentLastEventId();
  for (Action a: dataSource.getAllDocumentsBetween(currentCheckpointValue, currentLas
      if (a.kind == ADD) papi.addDocument(a.getDocument());
      else if (a.kind == DEL) papi.deleteDocument(a.getURI());
  }
  /* Now, set in CloudView the fact that we have reached currentLast */
  papi.setCheckpoint(currentLast, CHECKPOINT_NAME);
}
```

You can force the sync when you set a checkpoint, or just after, but this is not strictly necessary. If you don't sync and a crash occurs, you will retrieve the previous checkpoint value, and will re-scan more documents than needed. However, the indexing process is idempotent when the same document is pushed several times, therefore, there is no change to the database.

Synchronization best-practices

- In some cases, you will need to garbage-collect the data source log after pushing. Make sure in this case that you sync Exalead CloudView before garbage-collecting the log.
- Always compute the "next" checkpoint value before scanning. This way, if new records are added while scanning, you will not miss them.
- As much as possible, avoid using the current time as a checkpoint value because in some rare cases, it can cause synchronization issues. Consider the following cases:
  - t0: A new transaction begins on the data source.
  - t1: A document D1 is added on the transaction; its last modification date is set to "t1".
  - t2: Scan begins; we compute t2 as the next checkpoint value, and will scan the source up to t2.
  - t3: Transaction commits.

    At the end, the document D1 has not been scanned. However, we will resume the scan from t2 next time, and therefore never scan D1.
- If you don't have any other form of ever-increasing ids, and must use modification/current times, make sure to always include an "overlap offset" when computing the checkpoint, to account for currently running transactions. For example:

  ```
  long now = System.currentTime();
  String nextCheckpoint = "" + (now - 60000);
  // Leave one minute of overlap: we'll always scan a bit more than needed,
  // but won't miss any documents in currently running transactions.
  ```

Push API filters

The PushAPI class can be encapsulated using different Push API filters to enhance or modify its behavior. The resulting class inherits the PushAPI, allowing to replace the original one.

About Push API filters

Push API filters include buffering, logging capabilities, debugging features plus custom features.

Filters have generally one constructor taking a parent PushAPI object to override or enhance its features. Other constructors may be used to tune the default settings.

Push API filters must be threadsafe if the connector using it:

- Supports the fetch operation. The same PushAPI pipeline is used for both scan and fetch operations, which can occur concurrently.
- Declares itself as reentrant (`ConnectorCapabilities#canFetch`). There can be more than one scan at the same time.
- Uses a thread-pool to speed up the push of documents.

  Important: You cannot add Push API filters on the Push API of the Indexing server. It is however possible to use them in the Java Client code that sends documents.

Built-in classes

Push API filters include the following built-in classes:

Class	Description

Background PushAPI Filter
`com.exalead.papi.framework.connectc`
`BackgroundPushAPIComponent`
	Sends documents in the background. Use this filter when a lot of small files are sent to the PushAPI and slow it down considerably.

Buffering PushAPI Filter
`com.exalead.papi.framework.connectc`
`BufferedPushAPIComponent`
	Buffers PushAPI operations in memory, and executes them by batch.
	Example: if you launch ten `addDocument()` operations, this class will attempt to collate them into a single `addDocumentList()` operation.
	Caution: If the final `papi.sync()` method is not called by the last BufferedPushAPI, don't forget to force the indexing of pending

Class	Description
	operations with the `papi.sync()` method after the last `addDocument()` operation for each BufferedPushAPI. This will prevent documents from remaining in the buffer and not be indexed.
Convert PushAPI Filter `com.exalead.papi.framework.connectc ConvertPushAPIComponent`	Specifies the elements you want to convert from documents. You can choose a conversion mode, filter the type of document binary parts to include/exclude, or filter documents on their file names. The main parameter of this filter is **Conversion mode**: • **Text** - retrieves only the textual content of the document and adds it to the `text` meta. • **Metadata** - retrieves texts and metadata extracted from binary parts and maps them to the document. Note that by default, metadata is prefixed by `convert_` to distinguish it from the original document metadata. This prefix can be changed in the **Advanced Settings** if needed. • **Binary** - retrieves the result of the conversion as such in an Exalead (Ndoc) binary part that can be decoded using the **NativeTextExtractor** document processor in the analysis pipeline.
Disabled PushAPI Filter `com.exalead.papi.framework.connectc DisabledPushAPIComponent`	Does not send documents. Use this filter to test a connector without sending documents.
Dump PushAPI Filter `com.exalead.papi.framework.connectc DumpPushAPIComponent`	Dumps the documents being added to the PushAPI in logs, for debugging and audit purposes. The logs may include all metadata and fields sent through the PushAPI, attachments, etc.
Fake PushAPI Filter `com.exalead.papi.framework.connectc FakePushAPIComponent`	Simulates a fake remote Push API server. The parent Push API is unused, and all operations such as `addDocument()`, are emulated in memory, but no commands are transmitted to the remote Push API server.

Class	Description
	This is useful to perform tests on a connector, or to measure raw performance for the connector itself. This class is an enhanced version of the `DisabledPushAPI` class, as it emulates commands such as `enumerateCheckpointInfo()` or `enumerateSyncedEntries()` with already stored information.
Indexing Job Trigger Filter `com.exalead.papi.framework.connectc IndexTriggerPushAPIComponent`	This simple wrapper class sends a `triggerIndexingJob()` at the end of a session (`stopPushSession()`). Used by default for managed connectors.
Java PushAPI Filter `com.exalead.papi.helper.pipe.inline InlineJavaAPI`	Adds a Push API filter that can handle Java code. It takes Java code either inline or from a file, and executes it on-the-fly. For production mode, we recommend packaging custom code as a Java Plugin (CVPlugin) and referencing the path of the class file.
Metadata Compaction PushAPI Filter `com.exalead.papi.framework.connectc MetaCompactPushAPIComponent`	Serializes metas in an optimized compact format for the Push API. It is useful when documents have a lot of metas, as the PushAPI HTTP protocol is not efficient and the PushAPI server fetches metas one after the other.
Replay PushAPI Filter `com.exalead.papi.framework.connectc ReplayPushAPIComponent`	Adding this Push API filter is a prerequisite to use the Replay connector, which allows you to repush data from a given source. See "Replay Connector" in the *Exalead CloudView Connectors Guide* . Enter the Replay server name you defined previously as **Instance name** for the **Deployment > Roles > Data integration > Replay server** role.
Tee PushAPI Filter `com.exalead.papi.framework.connectc TeePushAPIComponent`	This wrapper duplicates all commands, and sends them to a secondary PushAPI. For debugging purpose only.

Class	Description

Tracing PushAPI Filter
`com.exalead.papi.framework.connectc`
`TracePushAPIComponent`
	This wrapper adds simple logging capabilities, recording regularly the number of documents sent, the bandwidth used etc.

Code snippet (Java)

```
PushAPI papi;
// original PushAPI
// Override the current papi with buffering capabilities.
// Documents passed to this new papi will be buffered.
papi = new BufferedPushAPI(papi);
// Then add logging capabilities. Documents passed to this new papi
// will first be recorded for logging and then batched.
papi = new TracePushAPI(papi);
```

Deploying Connectors on a Remote Server

We recommend deploying custom connectors as plugins within Exalead CloudView. You may however need to deploy them on a remote server and still want to benefit from the Exalead CloudView Administration Console to manage them.

The Exalead CloudView kit contains a set of JAR files that must be deployed on your remote server:

- `datainteg-java-commons.jar` - contains the Remote Scan Server (RScan). It allows to manage Exalead CloudView scan operations on remote connectors. To do so you must instantiate these connectors as described in the code sample below. Once the RScan server is deployed, you can create RScan Client connectors in the Exalead CloudView Administration Console to launch scan operations.
- Developing unmanaged connectors requires the following jar files:
  - `datainteg-java-commons.jar`
  - `cloudview-java-plugin.jar`
  - `papi-java-client.jar`
  - `papi-java-connector.jar`

Instantiate a connector

1. The following code snippet shows how to instantiate a connector (`myConnector`).

```
package com.exalead.papi.datainteg.connectors;
```

```
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import com.exalead.mercury.plugin.simple.SimplePluginManager;
import com.exalead.papi.datainteg.connectors.rscan.unmanaged.Runner;
import com.exalead.papi.framework.connectors.Connector;
import com.exalead.papi.framework.connectors.ConnectorConfig;
import com.exalead.papi.helper.Document;
import com.exalead.papi.helper.PushAPI;
public class RScanServerSample {
    public static void main(final String[] args) throws Exception {
        BasicConfigurator.configure();
        Logger.getRootLogger().setLevel(Level.INFO);
        SimplePluginManager.getCurrentInstance();
        // Start a RSCAN server
        final Runner rscanRunner = new Runner(10005);
        // Add a new connector
        rscanRunner.registerConnector(MyConnectorClass.class, "myConnector");
        try {
            // wait for input requests on RSCAN server indefinitely.
            Thread.sleep(Long.MAX_VALUE);
        } catch (final InterruptedException e) { /* do nothing */ }
        rscanRunner.stop();
    }
    /**
     * Connector sample pushing one document.
     */
    public static class MyConnectorClass extends Connector {
        public MyConnectorClass(final ConnectorConfig config) throws Exception {
            super(config);
        }
        @Override
        public void scan(final PushAPI papi, final String scanMode, final Object s
Exception {
            final Document doc = new Document("document1234");
            doc.addMeta("author", "foo");
            doc.addMeta("content", "Lorem ipsum dolor sit amet...");
            papi.addDocument(doc);
        }
    }
}
```

Launch your connector using a command line

1. Copy all .jar files from your Exalead CloudView `<INSTALLDIR>/sdk/java-customcode/lib` to a working directory.
2. Copy all the connector .jar files to the working directory.

3. Open a shell, go to your working directory and type the following command:

```
java -cp '*' com.exalead.papi.datainteg.connectors.rscan.server.RemoteScanLauncher
-rscanPort <port number> -connector <connector class>=<connector name>
```

where:

- `<port number>` is an available port (not used by CloudView).
- `<connector class>` is the name of your connector class, for example, `com.customer.cloudview.Connector1`
- `<connector name>` is the name of your connector.
- You can use the `-connector` parameter several times to launch several connectors at once.

The following snippet shows the source code for the `RemoteScanLauncher` class:

```
package com.exalead.papi.datainteg.connectors.rscan.server;
import java.util.ArrayList;
import java.util.List;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import com.exalead.mercury.plugin.simple.SimplePluginManager;
import com.exalead.papi.datainteg.connectors.rscan.unmanaged.Runner;
import com.exalead.papi.framework.connectors.Connector;
public class RemoteScanLauncher {
  private class ConnectorDescription {
    private String clazz;
    private String name;
    public ConnectorDescription(final String clazz, final String name) {
      this.clazz = clazz;
      this.name = name;
    }
    public String getClazz() {
      return clazz;
    }
    public String getName() {
      return name;
    }
  }
  private static final Logger logger = Logger
      .getLogger(RemoteScanLauncher.class);
  private int rscanPort = -1;
  private List<ConnectorDescription> connectors = new ArrayList<ConnectorDescripti
  private void parseCommandLine(final String[] args)
      throws CommandLineParameterException {
    if (args.length > 0) {
      int i = 0;
```

```
      while (i < args.length) {
        final String param = args[i++];
        if (param.equals("-rscanPort")) {
          if (args.length < i + 1) {
            throw new CommandLineParameterException("Missing " + param + " value")
          } else {
            final String rscanPortParam = args[i++];
            this.rscanPort = Integer.parseInt(rscanPortParam);
          }
        }
        else if (param.equals("-connector")) {
          if (args.length < i + 1) {
            throw new CommandLineParameterException("Missing " + param + " value")
          } else {
            final String connectorParam = args[i++];
            final String[] connector = connectorParam.split("=");
            if (connector.length != 2) {
              throw new CommandLineParameterException("Invalid " + param + " value
            }
            final ConnectorDescription connectorDescription = new ConnectorDescrip
connector[1]);
            connectors.add(connectorDescription);
          }
        }
        else {
          throw new CommandLineParameterException("Unknown command line parameter:
        }
      }
    }
  }
  private void validateCommandLineParameters()
      throws CommandLineParameterException {
    if (this.rscanPort == -1) {
      throw new CommandLineParameterException("Missing -rscanPort parameter");
    }
    if (this.connectors.size() == 0) {
      throw new CommandLineParameterException("Missing -connector parameter");
    }
  }
  private void displayHelp() {
    final String help = "Rscan connector launcher (c) Exalead\n"
        + "Usage: \n"
        + "1. Copy all .jar files from CLOUDVIEW/sdk/java-customcode/lib into your
        + "2. Copy all .jar files of your connector into your working directory\n"
        + "3. Into your working directory: java -cp '*'
com.exalead.papi.datainteg.connectors.rscan.server.RemoteScanLauncher -rscanPort <
 -connector <connector class>=<connector name>\n"
        + "   You can use several -connector parameters to launch several connecto
```

```
    logger.info(help);
  }
  private void run() throws Exception {
    SimplePluginManager.getCurrentInstance();
    // Start a RSCAN server
    final Runner rscanRunner = new Runner(this.rscanPort);
    try {
      // add new connectors
      for (final ConnectorDescription connector : connectors) {
        @SuppressWarnings("unchecked")
        final Class<? extends Connector> clazz = (Class<? extends Connector>)
Class .forName(connector.getClazz());
        final String name = connector.getName();
        rscanRunner.registerConnector(clazz, name);
        logger.info("Connector " + name + " is launched.");
     }
      Thread.sleep(Long.MAX_VALUE);
    } finally {
      rscanRunner.stop();
    }
  }
  public static void main(String[] args) throws Exception {
    BasicConfigurator.configure();
    Logger.getRootLogger().setLevel(Level.INFO);
    final RemoteScanLauncher launcher = new RemoteScanLauncher();
    try {
      launcher.parseCommandLine(args);
      launcher.validateCommandLineParameters();
      launcher.run();
    } catch (final CommandLineParameterException e) {
      logger.error(e);
      launcher.displayHelp();
    }
  }
}
```

Calculating a diff between Two Data Sources

It is sometimes useful to get the differences between two data sources.

First, you enumerate your data source and push all documents in the index using the Push API. Then, you need to regularly update the index by adding new documents, and deleting documents that have been deleted in the source.

Sometimes you are not notified by the source of deleted documents and you don't know which are the documents to delete from the index. In such case, your only solution is to compare documents

present in the index with documents present in the data source, and then delete documents that are in the index but no longer in the source.

To do so, we provide a `Subtractor` class in the `papi-java-datainteg-commons.jar` file.

Using this class you will create an object that calculates the set of items present in a data source A and NOT in a data source B, to know exactly which items to delete from the index.

This calculation will be performed in a java heap buffer and will swap on disk if there is not enough memory available. Items enumeration is processed through Cursor objects, which are enumerators. This allows you to use the `Subtractor` class with various sources, you just need to provide enumerators to access items.

```
/**
 * Iterator on the CloudView index
*/
private Cursor<byte[]> getCursorFromCloudview(final PushAPI papi) throws PushAPIExcep
    return new Cursor<byte[]>() {
        private Iterator<SyncedEntry> syncedEntries = papi.enumerateSyncedEntries("",
EnumerationMode.NOT_RECURSIVE_ALL).iterator();
                @Override public void close() throws IOException {
        }
                @Override
        public byte[] next() throws Exception {
            if( syncedEntries.hasNext() == false ) {
                return null;
            }
                        final SyncedEntry entry = syncedEntries.next();
            final String uri = entry.getUri();
                        return uri.getBytes("UTF-8");
        }
    };
}
/**
  * Iterator on a fake data source
  * which contains 9995 documents with uris from "uri0" to "uri9994"
  */
private Cursor<byte[]> getCursorFromDataSource() {
    return new Cursor<byte[]>() {
        private int index = 0;
        private int len = 9995;
        @Override
        public void close() throws IOException {
        }
        @Override
        public byte[] next() throws Exception {
            if( index >= len )
                return  null;
```

```
            final String uri = new String("uri" + index++);
            return uri.getBytes("UTF-8");
        }
    };
}
/**
  * Sample code to show how to check documents needed to be deleted in
  * the Cloudview index.
  * In this sample code we already have 10000 (10K) documents in the Cloudview index
  * with uris from "uri0" to "uri9999"
  */
private void checkItemsToDelete(final PushAPI papi, final Logger logger) throws Excep
    // computes sources intersection
    final File workdir = new File(System.getProperty("java.io.tmpdir") + '/' + UUID.r
         try {
        // this cursor will enumerate on 10000 documents (the cloudview index)
        final Cursor<byte[]> cursorFromCloudview = getCursorFromCloudview(papi);
                try {
            // this cursor will enumerate on 9995 documents (the data source)
            final Cursor<byte[]> cursorFromDataSource = getCursorFromDataSource();
                        try {
                // maximum amount of memory consumed in Java HEAP (5 MB)
                final int ramBudget = 5 * 1024 * 1024;
                  // create a subtractor object
                final Subtractor sub = new Subtractor(workdir, "subtractorName", ramB
                     // computes items that are in source1 and NOT in source2
                 final Cursor<byte[]> itemsInCloudviewAndNotInDataSource = sub.sub(cu
 cursorFromDataSource);
                                    try {
   // loop on all items that are in the Cloudview index and no longer in the data sou
                    for (final byte[] bytes : new IterableCursor<byte[]>(itemsInCloud
 {
                        final String s = new String(bytes, "UTF-8");
                        papi.deleteDocument(s);// delete the document to update the i
                    }
                }
                finally {
                    itemsInCloudviewAndNotInDataSource.close();
                }
            }
            finally {
                cursorFromDataSource.close();
            }
        }
        finally  {
            cursorFromCloudview.close();
        }
    }
```

```
    finally {
        FileUtils.deleteQuietly(workdir);
    }
}
```

Customizing Connectors to use the Interconnector Service

You can customize connectors to allow the use of the Interconnector service between them.

Required dependencies

You must first add the following JAR files located in `<DATADIR>/javabin/plugin` to your project:

- `interconnector-service-java-framework.jar`
- `datainteg-java-commons-queue.jar`

Master connector sample code

To allow connection between the connectors and the Interconnecter server, you must first check that an Interconnector server has been deployed in the Administration Console (**Deployment > Roles**). For more details, see "Configure the Interconnector Server" in Exalead CloudView Connectors Guide.

You must also add two configuration keys to your connector:

- the Interconnector server instance name
- the slave connector name

Below is a sample code for your master connector (JDBC here).

```
//Master connector
//While processing a column containing a path, adds a File System Query (FS Query = a
//Instantiation of the Interconnector Service
InterConnectorServiceBuilderImpl builder = (InterConnectorServiceBuilderImpl) InterCo
builder.withDestination(config.slaveConnector);  //a configuration key has been added
//name of the slave connector
builder.withQuerySerializer(new FileSystemQuerySerializer()); //the file system query
//the JDBC connector
builder.withInterconnectorServerInstance(config.interconnectorServerInstanceName); //
//added to the connector, to know the interconnector server instance name the query w
InterConnectorServiceImpl service = builder.build();    //this is time consuming, the
//only once per application (as a Singleton)
//End of the instantiation
//Creation of the File System Query
FileSystemQuery fileSystemQuery = new FileSystemQuery();
```

```
fileSystemQuery.setPath(filePath);
//Calls to the service to delete and add a query
service.deleteQuery(docURI.toString());  //clear the query before adding the new one
service.addQuery(fileSystemQuery, false, true, docURI.toString());   //docURI is the
that is currently processed
//Creation of the parent document in the Consolation Box, with type "aggregated"
PushAPITransformationHelpers.addArcTo(document, "parent", docURI.toString() + "_REL");
PushAPITransformationHelpers.setType(document, "aggregated");
//Don't forget to close the service when all the processing is done
service.closeService();
```

Slave connector sample code

Below is a sample code for your slave connector (File System here).

```
//Slave connector
//Enumerates the watched queries
 InterConnectorServiceBuilderImpl builder = (InterConnectorServiceBuilderImpl) InterC
 builder.withReceiverName(key.connector.getConnectorName());  //the receiver is the c
 builder.withQuerySerializer(new FileSystemQuerySerializer());
 builder.withInterconnectorServerInstance(config.interconnectorServerInstanceName);
 InterConnectorServiceImpl service = builder.build();  //this is time consuming, the
//once per application (as a Singleton)
 service.pollMessageQueue();
 Iterable<ImmutablePair<String, UserPayloadWithUri<String, String>>> tripletIterable
 Iterator<ImmutablePair<String, UserPayloadWithUri<String, String>>> iteratorQueries
  if (iteratorQueries != null && iteratorQueries.hasNext()){
     try {
         final ImmutablePair<String, UserPayloadWithUri<String, String>> triplet = it
         UserPayloadWithUri<String, String> queryAndFlags = triplet.getRight();
         Query query = service.getSerializer().deserialize(queryAndFlags.getValue());
         String checkpoint = triplet.getLeft();
         String filepath = query.getUID();
         ...
         FilesystemKey skey = new FilesystemKey(key.connector, filepath, connectorcon
(filesystemRootPathConfig), false, true);
         try {
  service.notifyEndOfQueryJob(checkpoint);
         }
         catch (Exception e){
  logger.warn("Error while notifying end of query job to storage");
         }
         return (FSKey) skey;
     }
     catch (Exception e){
         logger.error("Error while adding a root key ",e);
         return null;
     }
```

```
 }
 //Processes a watched query, i.e. a file system path in this connector
 //Adding a "parent_uri" meta to link the indexed file system document to the indexed
 try {
     ArrayList<String> listParentURIs = service.getParentURIFromUID(file.getAbsoluteP
     if (listParentURIs != null && !listParentURIs.isEmpty()){
         for (String parentURI : listParentURIs){
  collect.addMeta("parent_uri", parentURI);
         }
     }
     service.closeService();
 }
 catch (Exception e ){
     logger.debug("Error retrieving parent URI while building PAPI document "+ absolu
 }
 //Processes the "parent_uri" metas to create arcs and documents in the consolidation
  Collection<Meta> parents_meta = doc.getMetaContainer().getMetaValues("parent_uri");
  if (parents_meta != null &&  !parents_meta.isEmpty()){
      Iterator<Meta> iterator = parents_meta.iterator();
      while (iterator.hasNext()){
          String uri = iterator.next().getValue();
        // creating the "relation" intermediate document in the consolidation box, th
          PushAPITransformationHelpers.createUnmanagedDocument(doc, uri + "_REL",  "r
          PushAPITransformationHelpers.addArcFrom(doc, "rel",  uri + "_REL");
      }
      PushAPITransformationHelpers.setType(doc, "child");
  }
//Enumerates and processes the deleted queries
Iterable<ImmutablePair<String, UserPayload<String, String>>> deleteIterable = service
Iterator<ImmutablePair<String, UserPayload<String, String>>> iteratorDeletedQueries =
while (iteratorDeletedQueries != null && iteratorDeletedQueries.hasNext()){
         final ImmutablePair<String, UserPayloadWithUri<String, String>> triplet = it
         UserPayloadWithUri<String, String> queryAndFlags = triplet.getRight();
         Query query = service.getSerializer().deserialize(queryAndFlags.getValue());
         String filepath = query.getUID();
         papi.deleteDocument(filepath);
 }
```

Interconnector aggregation processor

You must now configure the Interconnector aggregation processor in the Administration Console with the appropriate document types and arcs defined in your code. For more details, see "Add the Interconnector aggregation processor" in Exalead CloudView Connectors Guide.

You can scan your master connector, then your slave connector.

Best Practices

Crash resistance

To test the connector crash resistance, you can:

- Stop the source server during indexing time to simulate a source server crash.
- Unplug the network cable to simulate a network error.
- Restart the Push API server while the connector is indexing to simulate a Push API server crash.

All these tests should pass without losing any document.

Log management

Exalead CloudView uses log4j to report logs. You can:

- either use the `getLogger()` static method in the `Logger` class,
- or the `getLogger()` method of the `Connector` object.

The global log level of the product is managed in the **Logs** menu of the Administration Console. You can:

- Display the exception stack for each message.
- Log the URIs of documents sent to the index in trace mode.
- Log the plugin version number at the beginning of the scan method.

  Note: You can also configure log levels more precisely by editing the `<DATADIR>/config/Logging.xml` file.

Test plan & monitoring

These are a few tests that you can perform to test your connector:

- Index 1 million documents in a single indexing phase without crash.
- Calculate the required time for incremental indexing just after a full scan, without any modification on the source server. This will give you an idea of the minimum time required for incremental indexing.
- Launch several incremental indexing and monitor memory consumption. Note that the connector process memory is shared by all connectors.

- If you encounter `java.lang.OutOfMemoryError: Java heap space` or `java.lang.OutOfMemoryError: PermGen space` errors in a specific process, the memory setting for this process may be too low.

  Edit `DeploymentInternal.xml`, and change the corresponding `<ProcessInternalConfig>` node value(s):

  - Change the `-Xmx` value for heap space issues. For example: `<StringValue value="-Xmx1024m"/>`
  - Change the `-XX:MaxPermSize` value for PermGen space issues. For example: `<StringValue value="-XX:MaxPermSize=1024m"/>`

    Do not forget to rebuild the configuration (for example with `<DATADIR>/bin/buildgct master`).

Package the connector

Do not forget to update the plugin version number for each new release.

Aggregate Documents

Sometimes, building a PAPI document is a really complex task, especially when you need to rebuild it entirely for an incremental update. For example, let's say that for a connector indexing emails, we want to create a single PAPI document for each email thread that aggregates all the emails of the thread. When a new email arrives in a thread, the connector must rebuild the entire document by aggregating all emails once again.

For this kind of situation, we recommend using the Consolidation Server. See the Exalead CloudView Consolidation Server Guide.

Other best practices

- Index raw documents without connector aggregation. If you want to perform aggregation, use the Consolidation Server. See the Exalead CloudView Consolidation Server Guide.
- Do not store anything on the hard drive, everything must be stored in Exalead CloudView.
- Build document URIs in a hierarchical way, for example, `/ROOT/FolderA/FolderB/DocumentA`, to be able to delete a whole folder content with only one call to the `deleteDocumentsRootPath()` method.
- If the indexing is multi-threaded, the number of threads must be configurable in the connector UI to adjust the server load.

- To send documents as batches to the indexing server, you can select the **Buffer operations** option in the Administration Console **> Connectors > Deployment > Push API** section. You don't need to develop your own buffering strategy, just rely on this option.